Electronic Appendix to the article:
"Evolution in group-structured populations can resolve the Tragedy of the Commons"
by Timothy Killingback, Jonas Bieri, and Thomas Flatt

Here we provide an annotated version of the source code, written in Java, which we have used to run evolutionary simulations of the public goods game (PGG). All questions concering technical details of the source code or simulations should be directly addressed to the author of the code: jonas.bieri@itsystems.ch

PublicGoodsGame.java:

```
package jonasbieri.publicgoodsgame;

import jonasbieri.tools.*;

import java.util.*;
import java.io.*;

class PublicGoodsGame extends Thread {

    private Properties prop;
    private boolean runFlag = false;

    protected void setProperties(Properties p) {
        this.prop = p;
    }

    protected void setRunFlag(boolean runFlag) {
        this.runFlag = runFlag;
    }

    protected boolean getRunFlag() {
        return runFlag;
    }

    public void run() {

        //parameters for simulation
        int groupCount;
        int indCount;
        long iterations;
```

```
double initialInvestmentFrom;
double initialInvestmentTo;
double mutationRate, mutationVariance;
double kFrom, kTo, kStep, v;
float dFrom, dTo, dStep;
long avgOfGenerationsCount; //# of generations that are used to calculate mean Investment at end

try {
    //get parameters for simulation from properties
    System.out.println("Reading Parameters");

    groupCount = Integer.parseInt(prop.getProperty("groupCount"));
    indCount = Integer.parseInt(prop.getProperty("individualCount"));
    iterations = Long.parseLong(prop.getProperty("iterations"));
    avgOfGenerationsCount = Long.parseLong(prop.getProperty("avgOfGenerationsCount"));
    initialInvestmentFrom = Double.parseDouble(prop.getProperty("initialInvestmentFrom"));
    initialInvestmentTo = Double.parseDouble(prop.getProperty("initialInvestmentTo"));
    mutationRate = Double.parseDouble(prop.getProperty("mutationRate"));
    mutationVariance = Double.parseDouble(prop.getProperty("mutationVariance"));
    System.out.println("mutation Rate +/- Variance: " + mutationRate + " +/- " + mutationVariance);

    kFrom = Double.parseDouble(prop.getProperty("kFrom"));
    kTo = Double.parseDouble(prop.getProperty("kTo"));
    kStep = Double.parseDouble(prop.getProperty("kStep"));
    v = Double.parseDouble(prop.getProperty("V"));

    dFrom = Float.parseFloat(prop.getProperty("dFrom"));
    dTo = Float.parseFloat(prop.getProperty("dTo"));
    dStep = Float.parseFloat(prop.getProperty("dStep"));
    System.out.println("dispersal rate from " + dFrom + " to " + dTo + " Step " + dStep);

    String disp = prop.getProperty("isDispersalStochastic");
    boolean isDispersalStochastic = false;
    if (disp.equals("true")) isDispersalStochastic = true;
    System.out.println("isDispersalStochastic:" + isDispersalStochastic);

    int simulations = Integer.parseInt(prop.getProperty("simulations"));
    if (simulations <= 0) throw new RuntimeException("at least 1 simulation must be run! specify in property file.");
    System.out.println("simulations:" + simulations);

    String details = prop.getProperty("showDetails");
    boolean showDetails = false;
    if (details.equals("true")) showDetails = true;
    System.out.println("ShowDetails:" + showDetails);
```

```
            //create output file
            System.out.println("Creating output File '" + prop.getProperty("outputFile") + "'...");
            PrintFile out = new PrintFile(prop.getProperty("outputFile"));
            //create headers
            if (showDetails) {
                out.println("sim \t size \t groups \t k \t V \t d \t stochasticDispersal \t mutRate \t mutVar \t generation
\t averageInvestment \t avgOfLastGenerationsCount \t floatingAverageInvestment");
            } else {
                out.println("sim \t size \t groups \t k \t V \t d \t stochasticDispersal \t mutRate \t mutVar \t generation
\t averageInvestment \t avgOfLastGenerationsCount \t floatingAverageInvestment");
            }

            //----HERE SIMULATION STARTS-----------
            for (double k = kFrom; k <= kTo; k += kStep) {
                for (float d = dFrom; d <= dTo; d += dStep) {
                    System.out.println("======= d " + d + "========");
                    for (int simCounter = 0; simCounter < simulations; simCounter++) {
                        //Start Simulation
                        System.out.println("Simulation Run #" + simCounter);
                        System.out.println("--------------");
                        System.out.println("Simulating " + iterations + " iterations...");
                        //create Individuals
                        Individual[] ind = new Individual[indCount];
                        for (int i = 0; i < ind.length; i++) {
                            ind[i] = new Individual(Zufall.simulate(initialInvestmentFrom, initialInvestmentTo),
mutationRate, mutationVariance, v, k);
                        }

                        //create new Population
                        Population p = new Population(ind);
                        p.createGroups(groupCount, k, v);
                        //generation number starting at which average Investment needs to be calculated
                        long startAveragingFromGeneration = iterations - 1 - avgOfGenerationsCount;
                        long generationsForAveragingCount = 0;
                        double sumOfMeanInvestments = 0;
                        //counter that counts how many iterations have been done
                        long iterationsCounter = 0;

                        //iteration loop
                        for (long i = 0; i < iterations - 1; i++) {
                            iterationsCounter = i;
                            if (showDetails) {
```

```
                                out.println(simCounter + "\t" + +indCount + "\t" + groupCount + "\t" + k + "\t" + v + "\t" +
d + "\t" + isDispersalStochastic + "\t" + mutationRate + "\t" + mutationVariance + "\t" + i + "\t" +
p.getAverageInvestment());
                            }



                            //control output on screen
                            if (i % 50 == 0) {
                                System.out.println(i + " " + p.getAverageInvestment() + "         " + sumOfMeanInvestments /
generationsForAveragingCount);
                            }

                            //zero investment: give out zero, not mean of last n generations
                            if (p.getAverageInvestment() == 0) {
                                sumOfMeanInvestments = 0;
                                generationsForAveragingCount = 1;
                                break;
                            }
                            p.makeGroupsPlay();

                            //give Gui a chance to react to user intervention
                            this.yield();
                            //check if Simulation should stop
                            if (!runFlag) break;

                            p.replicate();
                            p.mutateIndividuals();
                            p.disperse(d, isDispersalStochastic);

                            //check if average of investment already needs to be calculated
                            if (i >= startAveragingFromGeneration) {
                                generationsForAveragingCount++;
                                sumOfMeanInvestments += p.getAverageInvestment();
                            }
                        }
                        //parameters at end of simulation
                        double floatingMeanInvestment = sumOfMeanInvestments / generationsForAveragingCount;
                        out.println(simCounter + "\t" + indCount + "\t" + groupCount + "\t" + k + "\t" + v + "\t" + d + "\t"
+ isDispersalStochastic + "\t" + mutationRate + "\t" + mutationVariance + "\t" + (iterationsCounter - 1) + "\t" +
p.getAverageInvestment() + "\t" + avgOfGenerationsCount + "\t" + floatingMeanInvestment);

                        System.out.println("generationsforAveragingInvestmentsCount = " + generationsForAveragingCount);
```

```
                    //give Gui a chance to react to user intervention
                    this.yield();
                    //check if Simulation should stop
                    if (!runFlag) break;
                }
                //give Gui a chance to react to user intervention
                this.yield();
                //check if Simulation should stop
                if (!runFlag) break;
            }
        }

        System.out.println("Closing output file...");
        out.close();
        System.out.println("...done!");
    } catch (IOException e) {
        System.out.println(e);
        System.exit(0);
    }
}


public static void main(String[] args) {
    PublicGoodsGame p = new PublicGoodsGame();
    p.start();
}
}
```

Population.java:

```
package jonasbieri.publicgoodsgame;

import jonasbieri.tools.*;

class Population{

      //variables
      private int groupCount;
      private int size;
      private Group[] groups, newGroups;
      private Individual[] individuals;

      //methods
```

```
      //constructor
      protected Population(Individual[] individuals){
            if (individuals.length==0) throw new RuntimeException("You tried to create a Population with no Individuals in
it!");
            this.individuals=individuals;
            size=individuals.length;
            groupCount=0;
      }

/**************************************************************************
 * ACCESSORS                                                              *
 **************************************************************************/
////////////////////////////////////////////////////////////////////////////
// getSize:                                                                //
////////////////////////////////////////////////////////////////////////////
/*
* @return   int               number of individuals
*/
public int getSize()
{
  return individuals.length;
}//method

      public int getGroupCount(){
            return groups.length;
      }

      //create groups and attribute each individual to one group (sequentially)
      //replication produces a shuffled population, i.e. no shuffling needed at this step.
      protected void createGroups(int groupCount, double k, double v){
            this.groupCount=groupCount;
            this.groups=new Group[groupCount];

            //check if (i) population size  is >0, (ii) population size is dividable by groupCount, (iii) groupCount is >0
            if (groupCount<=0) throw new RuntimeException("At least one group must be created!");
            if (size<=0) throw new RuntimeException("No Individuals in Population to create Groups with!");
            if ((size%groupCount)!=0) throw new RuntimeException("Population Size must be dividable without rest by number of
Groups!");

            //create Groups
            for (int i=0; i<groupCount; i++){
                  groups[i]=new Group(k, v, i);
            }
```

```
            //go through Individuals, and add them to a group (sequentially)
            int groupSize=(individuals.length/groupCount);
            for (int i=0; i<individuals.length; i++){
                  groups[i/groupSize].addIndividual(individuals[i]);
            }

      }


      //disperse Individuals between groups.
      //parameters:
      //prop = proportion of Individuals dispersing (0<=prop<=1)
      //stochastic = if true, the number of Individuals emigrating from a group is a binomially distributed random number,
      //    if false, the number of emigrants is the same for all Groups, as determined by prop. Then the Groups have the same
size throughout the simulation

      protected void disperse(float prop, boolean stochastic){

            //create Group of dispersed Individuals (pool) with id -2
            Group pool=new Group(0,0,-2);
            int numberToPick=0;
            //go through Groups and remove randomly picked Individuals, put them into pool
            for (int i=0; i<groups.length; i++){
                  //determine number of Individuals to remove from Group (if Group is empty: none removed...)
                  if (stochastic){
                        numberToPick=Zufall.binomial(prop, groups[i].getSize());
                  }
                  else{
                        numberToPick=Math.round(prop*groups[i].getSize());
                  }
                  //remove from Group, put into pool
                  for (int j=0; j<numberToPick; j++){
                        pool.addIndividual(groups[i].removeRandomIndividual());
                  }
            }

        //add Individuals from pool to Groups
        if (stochastic) {
            //add randomly chosen Individual from pool to randomly chosen Group
            int poolSize = pool.getSize();
            for (int i = 0; i < poolSize; i++) {
                int randomGroup = (int) Zufall.simulate(0, groups.length - 1);
                groups[randomGroup].addIndividual(pool.removeRandomIndividual());
```

```
        }
    } else { //deterministic dispersal
        //every group gets the same amount of individuals from the pool
        //iterate over groups, add an individual from pool to group before going to next group
        int i=0;
        while (pool.getSize() > 0){
            //define which group gets an individual now
            i = i % groups.length;
            groups[i].addIndividual(pool.removeRandomIndividual());
            //make sure the next individual goes to the next group
            i++;
        }
    }



        //check if pool is empty now: should be!
        if (pool.getSize()!=0) throw new RuntimeException("After dispersing: pool is not empty: "+pool.getSize());

}



//replicate individuals according to their score
protected void replicate(){

        //create new empty Groups
        newGroups=new Group[groups.length];
        for (int i=0; i<groups.length; i++){
              newGroups[i] = new Group(groups[i].getK(), groups[i].getV(), groups[i].getGroupId());
        }
        //create array containing relative fitnesses
        double[] fit = new double[size];
        //populate the array with the scores
        //first: find smallest score in pop (may be negative). add (minScore+Double.Min_Value) to all scores
        //else: Individuals with score <=0 would NEVER be picked!
        double minScore=Double.MIN_VALUE;
        for (int i=0; i<individuals.length; i++){
              if (individuals[i].getPayoff()<minScore)
                    minScore=individuals[i].getPayoff();
        }
        if (minScore>=0) {
              minScore=0;
```

```
        }
        else{
              minScore=-minScore;
        }

        fit[0] = individuals[0].getPayoff()+minScore+Float.MIN_VALUE;
        for(int i=1; i<size; i++){
          fit[i] = fit[i-1] + individuals[i].getPayoff()+minScore+Float.MIN_VALUE;
        }

        //create new array of individuals
        Individual[] nextGeneration=new Individual[size];
        //determine individuals of next generation in relation to calculated fitnesses
        for(int i=0; i<size; i++) {
          double rand = Zufall.simulate(0.0d, fit[size-1]);
          int j = 0;
          while(rand > fit[j]) {
            j++;
          }
          try{
                    //clone Individual
                    nextGeneration[i]=(Individual)individuals[j].clone();
                    //add new Individual to parent's Group
                    int newGroupID=nextGeneration[i].getGroup().getGroupId();
                    newGroups[newGroupID].addIndividual(nextGeneration[i]);
          }
          catch (CloneNotSupportedException e){
                    throw new RuntimeException("cloning of Individuals not supported!");
          }
          //resetPayoff of nextGeneration Individual
          nextGeneration[i].resetPayoff();
        }
        //replace old array of Individuals
        individuals=nextGeneration;
        //replace old array of Groups
        groups=newGroups;

  }

  //make individuals within each group play with each other,
//but only if more than 1 individual is within each group
  protected void makeGroupsPlay(){
                for (int i=0; i<groupCount; i++){
                    if (groups[i].getSize() > 1)
```

```
                    {
                        groups[i].makeIndividualsPlay();
                    }
                }
        }

        //tell each Individual to mutate according to its probability and mutation size
        protected void mutateIndividuals(){
              for (int i=0; i<this.size; i++){
                    individuals[i].mutate();
              }
        }

        //calculate average investment of all individuals
        protected double getAverageInvestment(){
              double sumOfInvestments=0;
              for (int i=0; i<groupCount; i++){
                    sumOfInvestments+=groups[i].getSumOfInvestments();
              }
              return (sumOfInvestments/this.size);
        }

        //calculate average score of all individuals
        protected double getAverageScore(){
              double sumOfScores=0;
              for (int i=0; i<groupCount; i++){
                    sumOfScores+=groups[i].getSumOfScores();
              }
              return (sumOfScores/this.size);
        }


        public String toString(){
              String info="pop: size"+size;
              for (int i=0; i<groups.length; i++){
                    info+="group"+i+":"+groups[i];
              }
              return info;
        }

}
```

Group.java:

```
////////////////////////////////////////////////////////////////////////////
// Class: Group.java                                                      //
////////////////////////////////////////////////////////////////////////////
package jonasbieri.publicgoodsgame;

import java.util.*;
import jonasbieri.tools.*;

/**
 * A set of individuals. Individuals only interact with other members of the
 * group.
 *
 * @author Jonas Bieri
 * @author Rahel Luethy
 */

class Group{
////////////////////////////////////////////////////////////////////////////
// PRIVATE GROUP ATTRIBUTES                                               //
////////////////////////////////////////////////////////////////////////////
/**
 * unique identifier of a group.
 */
private int         id = -1;

/**
 * member individuals of a group.
 */
private LinkedList  individuals;

/**
 * sum of all investments of a groups member individuals.
 */
private double      sumOfInvestments  = 0;

/**
 * determines the maximal money each individual owns
 */
private double      k;

/**
 * curvature of hyperbolic [group component] function.
 */
```

```
private double      v;

////////////////////////////////////////////////////////////////////////////
// CONSTRUCTORS                                                           //
////////////////////////////////////////////////////////////////////////////
protected Group(double k, double v, int id)
{
  this.k     = k;
  this.v     = v;
  this.setGroupId(id);
  individuals = new LinkedList();
}//constructor

protected Group(Individual[] individuals,
                double k,
                double v,
                int id)
{
  this(k, v, id);
  this.individuals = new LinkedList(Arrays.asList(individuals));

  for (Iterator iterator = this.individuals.iterator(); iterator.hasNext();)
  {
    Individual individual = (Individual)iterator.next();
    individual.setGroup(this);
  }
}//constructor

/****************************************************************************
* METHODS                                                                   *
****************************************************************************/
////////////////////////////////////////////////////////////////////////////
// makeIndividualsPlay:                                                   //
////////////////////////////////////////////////////////////////////////////
  /**
   * make Individuals of the group play with each other.
   *
   */
  protected void makeIndividualsPlay()
  {
    //n-player game
    //go through collection of Individuals
    //for every Individual: call method
    //Individual.nPlayerGameWithSharedInvestments
```

```
    Iterator it = individuals.iterator();
    while (it.hasNext())
    {
      ((Individual) it.next()).nPlayerGameWithSharedInvestments();
    }
  }//method


////////////////////////////////////////////////////////////////////////////
// toString:                                                              //
////////////////////////////////////////////////////////////////////////////
/**
 * Write state of a group as string representation.
 *
 * @return                      description of this group.
 */
public String toString()
{
  String info = "size=" + individuals.size();
  Iterator i = individuals.iterator();
  while (i.hasNext())
  {
    info += ("Ind:" + i.next());
  }
  return info;
}//method

/*****************************************************************************
* ACCESSORS                                                                  *
*****************************************************************************/
////////////////////////////////////////////////////////////////////////////
// getGroupId:                                                            //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                      unique identifier of this group.
 */
public int getGroupId()
{
  return id;
}//method

////////////////////////////////////////////////////////////////////////////
// getSize:                                                               //
////////////////////////////////////////////////////////////////////////////
```

```
/**
 * @return                       size of this group (i.e. number of member
 *                               individuals).
 */
public int getSize()
{
  return individuals.size();
}//method

////////////////////////////////////////////////////////////////////////////
// getK:                                                                  //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                       assymptote of hyperbolic [group component]
 *                               function.
 */
public double getK()
{
  return this.k;
}//method

////////////////////////////////////////////////////////////////////////////
// getV:                                                                  //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                       curvature of hyperbolic [group component]
 *                               function.
 */
public double getV()
{
  return v;
}//method

////////////////////////////////////////////////////////////////////////////
// getSumOfInvestments:                                                   //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                       sum over investments of all group members.
 */
public double getSumOfInvestments()
{
  sumOfInvestments = 0;

  Iterator it = individuals.iterator();
```

```
  while(it.hasNext())
  {
    sumOfInvestments+=((Individual)it.next()).getInvestment();
  }//while
  return sumOfInvestments;
}//method

////////////////////////////////////////////////////////////////////////////
// getAverageInvestment:                                                  //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                      mean investment of a member of this group.
 */
public double getAverageInvestment()
{
  return (double)getSumOfInvestments()/getSize();
}//method

////////////////////////////////////////////////////////////////////////////
// getSumOfScores:                                                        //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                      sum over scores (=fitness; [group component] +
 *                              [individual payoff]) of all group members.
 */
public double getSumOfScores()
{
  double sumOfScores = 0;
  Iterator it = individuals.iterator();
  while(it.hasNext())
  {
    sumOfScores+=((Individual)it.next()).getPayoff();
  }//while
  return sumOfScores;
}//method




////////////////////////////////////////////////////////////////////////////
// getIndividuals:                                                        //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                     individuals belonging to this group.
```

```
 */
public Individual[] getIndividuals()
{
  Individual[] inds = new Individual[individuals.size()];
  return (Individual[])individuals.toArray(inds);
}//method


/*****************************************************************************
* MUTATORS                                                                   *
*****************************************************************************/
//////////////////////////////////////////////////////////////////////////////
// setGroupId:                                                              //
//////////////////////////////////////////////////////////////////////////////
/**
 * @param  id                   unique identifier of a group.
 */
private void setGroupId(int id)
{
  this.id = id;
}//method

//////////////////////////////////////////////////////////////////////////////
// addIndividual:                                                           //
//////////////////////////////////////////////////////////////////////////////
/**
 * @param  individual           to be added to the group.
 * @return <code>true</code>    if list of individuals changed as consequence of
 *                              call.
 *         <code>false</code>   otherwise.
 */
protected boolean addIndividual(Individual individual)
{
  individual.setGroup(this);
  sumOfInvestments += individual.getInvestment();
  return individuals.add(individual);
}//method

//////////////////////////////////////////////////////////////////////////////
// removeRandomIndividual:                                                  //
//////////////////////////////////////////////////////////////////////////////
/**
 * remove a randomly picked Individual from group this group.
 *
```

```
 * @return                      randomly picked individual that was removed.
 */
protected Individual removeRandomIndividual()
{
  if (this.getSize() <= 0)
  {
    throw new RuntimeException(
                "Tried to remove an Individual from an empty Group!");
  }//if

  int z = (int)Zufall.simulate(0, this.getSize() - 1);
  Individual i = (Individual)individuals.remove(z);
  sumOfInvestments -= i.getInvestment();
  return i;
}//method


}//class
```

Individual.java:

```
////////////////////////////////////////////////////////////////////////////
// Class: Individual.java                                                 //
////////////////////////////////////////////////////////////////////////////
package jonasbieri.publicgoodsgame;

import jonasbieri.tools.*;

/**
 * The reproductive and interacting unit of a population. Each individual
 * belongs to a group.
 *
 * @author Rahel Luethy
 * @author Jonas Bieri
 */

class Individual implements Cloneable {
////////////////////////////////////////////////////////////////////////////
// PRIVATE CLASS ATTRIBUTES                                               //
////////////////////////////////////////////////////////////////////////////
    /**
     * [individual component] of this individual.
     */
```

```
    private double payoff;

    /**
     * how much this individual invests.
     */
    private double investment;

    /**
     * probability for a mutation (per invocation of <code<mutate()</code>).
     */
    private double mutRate;

    /**
     * money that each individual has available to spend
     */
    private double v;

    /**
     * constant that defines the ratio of money in the jackpot that is returned
     * to the individuals: k / n is returned to each individual, where n is
     * the group size. 1 <= k <= n.
     */
    private double k;

    /**
     * variance of the normal distribution from which the investment of a mutated Individual is drawn
     */
    private double mutVariance;

    /**
     * group this individual belongs to.
     */
    private Group group;

    /**
     * <em>testing only</em>. <p>How many times this individual acted as a playing
     * partner.
     */
    private int playCounter;

////////////////////////////////////////////////////////////////////////////
// CONSTRUCTOR                                                            //
////////////////////////////////////////////////////////////////////////////
    protected Individual(double investment,
```

```
                        double mutRate,
                        double mutVariance,
                        double v,
                        double k) {
        this.investment = investment;
        this.mutRate = mutRate;
        if (mutVariance <= 0)
            throw new RuntimeException(
                    "Tried to Create Individual " +
                    "with MutationVariance <=0: " + mutVariance);
        this.mutVariance = mutVariance;
        this.k = k;
        this.v = v;
        this.payoff = 0;
        this.playCounter = 0;
    }//constructor

    /**************************************************************************
     * METHODS                                                                *
     **************************************************************************/
////////////////////////////////////////////////////////////////////////////
// clone:                                                                 //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return clone               of this Individual.
     */
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }//method

////////////////////////////////////////////////////////////////////////////
// mutate:                                                                //
////////////////////////////////////////////////////////////////////////////
    /**
     * mutate with the given probability and size.
     */
    protected void mutate() {
        //decide whether to mutate
        if (Zufall.simulate() < mutRate) {
            //calculate size of mutation and mutate
            investment = Zufall.normal(investment, Math.sqrt(mutVariance * investment));
            //no negative investments allowed
            if (investment < 0) investment = 0;
            //no investment bigger than v allowed
```

```
            if (investment > v) investment = v;
        }//if
    }//method


////////////////////////////////////////////////////////////////////////////
// nPlayerGameWithSharedInvestments:                                      //
////////////////////////////////////////////////////////////////////////////
    /**
     * investments are put into jackpot and shared among all Individuals within a
     * group.
     */
    protected void nPlayerGameWithSharedInvestments() {
        //payoff must be zero before interacting with all Individuals
        if (payoff != 0) {
            throw new RuntimeException("n-player game: payoff is not zero " +
                    "before playing with all Individuals. " +
                    "payoff=" + payoff);
        }//if

        //helper variables to make the formula easier readable
        int n = this.getGroup().getSize();
        double sumOfInvestments = this.getGroup().getSumOfInvestments();

        //payoff = V - myInvestment + (k/groupSize)(sumOfInvestmentsWithinMyGroup)
        //--> myInvestment is taken into account

        payoff = v - this.getInvestment()
                + (k / n) * sumOfInvestments;

    }//method


////////////////////////////////////////////////////////////////////////////
// toString:                                                              //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return String               representation of this Individual.
     */
    public String toString() {
        return ("i=" + getInvestment() + ", s=" + getPayoff());
    }//method
```

```
    /*************************************************************************
     * ACCESSORS                                                             *
     *************************************************************************/
////////////////////////////////////////////////////////////////////////////
// getPayoff:                                                               //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return payoff               i.e. [individual component] of this individual.
     */
    public double getPayoff() {
        return this.payoff;
    }//method


////////////////////////////////////////////////////////////////////////////
// getInvestment:                                                           //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return investment           of this individual.
     */
    protected double getInvestment() {
        return investment;
    }//method

////////////////////////////////////////////////////////////////////////////
// getGroup:                                                                //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return group                this individual belongs to.
     */
    protected Group getGroup() {
        return group;
    }//method


////////////////////////////////////////////////////////////////////////////
// getMutationRate:                                                         //
////////////////////////////////////////////////////////////////////////////
    /**
     * @return the mutation rate.
     */
    public double getMutationRate() {
        return this.mutRate;
    }//method
```

```
    /*************************************************************************
     * MUTATORS                                                              *
     *************************************************************************/
////////////////////////////////////////////////////////////////////////////
// resetPayoff:                                                           //
////////////////////////////////////////////////////////////////////////////
    /**
     * set individual's payoff to <0>.
     */
    protected void resetPayoff() {
        this.payoff = 0;
    }//method

////////////////////////////////////////////////////////////////////////////
// setGroup:                                                              //
////////////////////////////////////////////////////////////////////////////
    /**
     * @param group                the group the individual belongs to.
     */
    protected void setGroup(Group group) {
        this.group = group;
    }//method


}//class
```

IndividualData.java:

```
////////////////////////////////////////////////////////////////////////////
// Class: IndividualData.java                                             //
////////////////////////////////////////////////////////////////////////////
package jonasbieri.publicgoodsgame;

/**
 * A singleton wrapper for data common to all individuals.
 *
 * @author Rahel Luethy
 */

public class IndividualData
{
```

```
////////////////////////////////////////////////////////////////////////////
// PRIVATE CLASS ATTRIBUTES                                               //
////////////////////////////////////////////////////////////////////////////
/**
 * single instance of this class.
 */
private static IndividualData instance;

/**
 * parameter k defining the ratio of jackpot that is returned to the individuals
 */
private double k;

/**
 * defines the total amount of money that each individual owns before playing
 */
private double v;

/**
 * probability to for mutation {@see Individual#mutate()}
 */
private double mutationRate;

////////////////////////////////////////////////////////////////////////////
// CONSTRUCTOR                                                            //
////////////////////////////////////////////////////////////////////////////
/**
 * a private constructor to guarantee singleton pattern.
 */
private IndividualData()
{
}//constructor

/*****************************************************************************
* ACCESSORS                                                                  *
*****************************************************************************/
////////////////////////////////////////////////////////////////////////////
// getInstance:                                                           //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                      an instance of this singleton.
 */
public static IndividualData getInstance()
{
```

```
  //lazy instanciation
  if (instance == null)
  {
    instance = new IndividualData();
  }
  return instance;
}//method

////////////////////////////////////////////////////////////////////////////
// getK:                                                                  //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                       K
 */
public double getK()
{
  return this.k;
}//method

////////////////////////////////////////////////////////////////////////////
// getV:                                                                  //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                       V
 */
public double getV()
{
  return this.v;
}//method


////////////////////////////////////////////////////////////////////////////
// getMutationRate:                                                       //
////////////////////////////////////////////////////////////////////////////
/**
 * @return                      probability of a mutation on invocation of
 *                              {@link Individual#mutate() mutate}.
 */
public double getMutationRate()
{
  return this.mutationRate;
}//method
```

```
/*****************************************************************************
* MUTATORS                                                                   *
*****************************************************************************/
//////////////////////////////////////////////////////////////////////////////
// setK:                                                                    //
//////////////////////////////////////////////////////////////////////////////
/**
 * @param  k Jackpot multiplication factor
 */
public void setK(double k)
{
  this.k = k;
}//method

//////////////////////////////////////////////////////////////////////////////
// setV:                                                                    //
//////////////////////////////////////////////////////////////////////////////
/**
 * @param  v                 sum of money of individual
 */
public void setV(double v)
{
  this.v = v;
}//method


//////////////////////////////////////////////////////////////////////////////
// setMutationRate:                                                         //
//////////////////////////////////////////////////////////////////////////////
/**
 * @param mutationRate          probability of a mutation on invocation of
 *                              {@link Individual#mutate() mutate}.
 */
public void setMutationRate(double mutationRate)
{
  this.mutationRate = mutationRate;
}//method

}//class
```

Gui.java:

```
package jonasbieri.publicgoodsgame;
```

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.net.URL;

public class Gui extends JFrame {

    private JPanel dataPanel, buttonPanel;
    private Mediator med;
    private boolean runFlag = true;

    public Gui() {

        this.setTitle("The Public Goods Game");

        Container p = this.getContentPane();
        p.setLayout(new BorderLayout());


        dataPanel = new JPanel(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        c.weightx = c.weighty = 1;
        buttonPanel = new JPanel(new FlowLayout());

//--------------------------------//
        Object[][] components = new Object[][]
        {
            {"Number of Groups: ", "groupCount", "15"},
            {"Number of individuals: ", "individualCount", "150"},
            {"kFrom (Jackpot Multiplication): ", "kFrom", "0"},
            {"kTo: ", "kTo", "3"},
            {"kStep: ", "kStep", "0.1"},
            {"v (Initial Money for each Individual): ", "V", "5"},
            {"dFrom: ", "dFrom", "0.0"},
            {"dTo: ", "dTo", "1.0"},
            {"dStep: ", "dStep", "0.1"},
            {"Is Dispersal Stochastic? ", "isDispersalStochastic", "false"},
            {"Number of Simulations: ", "simulations", "100"},
            {"Number of iterations: ", "iterations", "9999"},
            {"Output File: ", "outputFile", "Simulations.txt"},
            {"Show Details? ", "showDetails", "false"},
            {"Show Average of last N generations: ", "avgOfGenerationsCount", "1000"},
```

```
            {"Mutation Rate: ", "mutationRate", "0.01"},
            {"Relative Mutation Variance: ", "mutationVariance", "1"},
            {"Initial Investment From: ", "initialInvestmentFrom", "2"},
            {"Initial Investment To: ", "initialInvestmentTo", "2.1"},
        };

        for (int row = 0; row < components.length; row++) {
            //create label
            c.gridy = row;
            c.gridx = 0;
            c.anchor = c.LINE_START;
            c.insets = new Insets(0, 5, 0, 0);
            dataPanel.add(new JLabel(components[row][0].toString()), c);
            //create Text Field
            c.gridy = row;
            c.gridx = 1;
            Component comp = null;
            if (components[row][2] instanceof Boolean) {
                comp = new JCheckBox("", Boolean.getBoolean(components[row][2].toString()));
            } else {
                comp = new JTextField(components[row][2].toString(), 20);
            }
            comp.setName(components[row][1].toString());
            dataPanel.add(comp, c);
        }

//--------------------------------//


        med = new Mediator();
        med.registerGui(this);

        StartStopButton ssb = new StartStopButton("Start", med);
        buttonPanel.add(ssb);
        med.init();

        JScrollPane scrollPane = new JScrollPane(dataPanel);
        p.add(scrollPane, BorderLayout.CENTER);
        p.add(buttonPanel, BorderLayout.SOUTH);
        //p.validate();
    }


    protected boolean getRunFlag() {
```

```
    return this.runFlag;
}

protected void setRunFlag(boolean flag) {
    this.runFlag = flag;
}


public Properties getProperties() {
    Properties prop = new Properties();

    //collect contents of all Components
    Collection l = new LinkedList();

    //go through array. if Component is a JTextField, add it to the Collection
    Component[] cmp = dataPanel.getComponents();
    for (int j = 0; j < cmp.length; j++) {
        if (cmp[j] instanceof JTextField) {
            l.add(cmp[j]);
        }
    }
    //put all Components into an Array
    JTextField[] textFields = new JTextField[l.size()];
    textFields = (JTextField[]) l.toArray(textFields);


    //go through all JTextFields of JPanel, add a property to Properties for each one
    for (int k = 0; k < textFields.length; k++) {
        if (textFields[k] instanceof JTextField) {
            prop.setProperty(textFields[k].getName(), textFields[k].getText());
        }
    }
    System.out.println("Parameters: " + prop);
    return prop;
}


public static void main(String[] args) {


    JFrame f = new Gui();
    f.setSize(500, 500);
    f.setResizable(false);
    f.show();
```

```
        f.addWindowListener(
                new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                        System.exit(0);
                    }
                }
        );
    }
}
```

Mediator.java:

```
/*
 * Mediator.java
 *
 * Created on 30. Juni 2001, 14:36
 */

/**
 *
 * @author  jonas.bieri
 * @version
 */
package jonasbieri.publicgoodsgame;

public class Mediator {

    private StartStopButton ssb;
    private PublicGoodsGame pgg;
    private boolean pggRunFlag=false;
    private Gui gui;

    /** Creates new Mediator */
    public Mediator() {
        pgg=new PublicGoodsGame();
    }

    public void registerGui(Gui gui){
        this.gui=gui;
    }

    public void registerStartStopButton(StartStopButton ssb){
        this.ssb=ssb;
```

```
    }

    public void startStopButtonPressed(){
        //depending on run flag of PublicGoodsGame:
        //start or stop
        if (pggRunFlag){
            //Stop Simulation
            //change caption of StartStopButton
            ssb.setText("Start");
            System.out.println("Pressed Stop Button");
            //change runFlag of PublicGoodsGame
            pggRunFlag=!pggRunFlag;
            pgg.setRunFlag(pggRunFlag);
        }
        else {
            //Start Simulation
            //change caption of StartStopButton
            ssb.setText("Stop");
            System.out.println("Pressed Start Button");
            //change runFlag of PublicGoodsGame
            pggRunFlag=true;

            //start new pgg
            pgg=new PublicGoodsGame();
            //hand over Properties to PublicGoodsGame
            pgg.setProperties(gui.getProperties());
            pgg.setRunFlag(pggRunFlag);
            pgg.start();
        }

    }

    public void init(){
        ssb.setText("Start");
        pggRunFlag=false;
    }
}
```

StartStopButton.java:

```
/*
 * StartStopButton.java
 *
```

```
 * Created on 6. Juli 2001, 19:03
 */

package jonasbieri.publicgoodsgame;
import javax.swing.*;
import java.awt.event.*;
import jonasbieri.tools.*;

/**
 *
 * @author  jonas.bieri
 * @version 1.0
 */

public class StartStopButton extends JButton implements ActionListener{

    Mediator med;

    /** Creates new StartStopButton */
    public StartStopButton(String caption, Mediator med) {
        super(caption);
        this.med=med;
        addActionListener(this);
        med.registerStartStopButton(this);
    }

    public void actionPerformed(ActionEvent e){
        med.startStopButtonPressed();
    }
}
```

Zufall.java:

```
package jonasbieri.tools;

import java.util.*;
import java.io.*;

public class Zufall{

      private static Random r=new Random();

      /**
```

```
*@param s seed @link java.util.Random
*/
public static void setSeed(long s){
      r=new Random(s);
}


/**Generates Random float in Interval [L,U[
* @param L lower limit of Interval
* @param U upper limit of Interval
*/
public static float simulate(float L,float U){
      return L+((U-L)*r.nextFloat());
}

/**Generates Random double in Interval [L,U[
* @param L lower limit of Interval
* @param U upper limit of Interval
*/
public static double simulate(double L,double U){
      return L+((U-L)*r.nextDouble());
}


/**Generates Random long in Interval [L,U]
* @param L lower limit of Interval
* @param U upper limit of Interval
*/
public static long simulate(long L, long U){
      return Math.round(L+((U-L)*r.nextDouble()));
}


/**Generates Random Number in Interval [0,1[
*/
public static double simulate(){
      return r.nextDouble();
}


/**Generates normally distributed Random number
* @param m mean
* @param sd standard deviation
*/
```

```
    public static double normal(double m,double sd){
          return ((double)(r.nextGaussian()*sd)+m);
    }

public static int poisson(double mean){
    double sum;
      int ctr;
    float randnumb;

  if (mean>0){
    sum=0;
    ctr=-1;
    do {
      ctr+=1;
      do {
        randnumb=r.nextFloat();
      }
      while(randnumb<=0);
      sum-=(Math.log(randnumb)/mean);
    }
    while(sum<=1);
    return Math.abs(ctr);
  }
  else return 0;
 }


//COF[0]=76.18009172947146;
    //COF[1]=-86.50532032941677;
    //COF[2]=24.01409824083091;
    //COF[3]=-1.231739572450155;
    //COF[4]=0.1208650973866179e-2;
    //COF[5]=-0.5395239384953e-5;

private static double gammLn(double _x){
          final double COF[]={    76.18009172947146,
                                                                  -86.50532032941677,
                                                                  24.01409824083091,
                                                                  -1.231739572450155,
                                                                  0.1208650973866179e-2,
                                                                  -0.5395239384953e-5};
          double x,y,tmp,ser;
```

```
      y=_x;
      x=_x;
      tmp=x+5.5;
      tmp-=(x+0.5)*Math.log(tmp);
      ser=1.000000000190015;
      for (int j=0; j<=5; j++){
            y+=1;
      ser+=COF[j]/y;
      }
      return (-tmp+Math.log(2.5066282746310005*ser/x));
}



public static int binomial(float _p, int n){

      int nOld = (-1);
      double am,em,g,angle,p,bn1,sq,t,y;
      double pOld=(-1.0), pc,plog,pclog,en,oldg;

  //to satisfy compiler
      am=em=g=angle=p=bn1=sq=t=y=pc=plog=pclog=en=oldg=0;

      p=(_p<=0.5 ? _p : 1.0-_p);
      am=n*p;
      if (n<25){
            bn1=0.0;
            for (int j=1;j<=n;j++)
                  if (r.nextFloat()<p) ++bn1;
      }
      else{
            if (am<1.0){
                  g=Math.exp(-am);
                  t=1.0;
                  int j;
                  for (j=0; j<n; j++){
                        t*=r.nextFloat();
                        if (t<g) break;
                  }
                  bn1=(j<=n ? j:n);
            }
            else{
                  if (n !=nOld){
```

```
                        en=n;
                        oldg=gammLn(((float)en+1.0));
                        nOld=n;
                  }
                  if (p !=pOld){
                        pc=1.0-p;
                        plog=Math.log(p);
                        pclog=Math.log(pc);
                        pOld=p;
                  }
                  sq=Math.sqrt(2.0*am*pc);
                  do{
                        do{
                              angle=Math.PI*r.nextFloat();
                              y=Math.tan(angle);
                              em=sq*y+am;
                        } while ((em<0.0) || (em>=(en+1.0)));
                        em=Math.floor(em);
                        t=1.2*sq*(1.0+y*y)*Math.exp(oldg-gammLn(em+1.0)
                        -gammLn(en-em+1.0)+em*plog+(en-em)*pclog);
                  } while (r.nextFloat() >t);
                  bn1=em;
            }

      }
      if (p != (_p)) bn1=n-bn1;
      return (int)bn1;
}

//Albertos c++ binomial method
public static int binomial2(double p, int n){
      double      sum=0,
              p0=Math.pow((1-p),n),
              pk;
      int               k=-1;

      if ((p<0)||(p>1)){
      System.out.println("0<p<1 required");
      System.exit(0);
      }
      while (sum<r.nextFloat()){
            k++;
            pk=p0;
            if (k>0){
```

```
                        for (int m=1; m<=k; m++){
                              pk*=((n-m+1)/(double)m)*(p/(1-p));
                        }
                        sum+=pk;
                  }
            }
            return k;
  }
}
```

PrintFile.java:

```
package jonasbieri.tools;
import java.io.*;

/** Erleichtert Gebrauch von normalen Text-Files
*   @author Rahel Luethy
*   @version 11.99
*/

public class PrintFile extends PrintStream {

      /**  @param filename Filename als String*/
      public PrintFile(String filename)
      throws IOException {
      super(
            new BufferedOutputStream(
                  new FileOutputStream(filename)));
      }

      public PrintFile(File file)
      throws IOException {
            this(file.getPath());
      }
}
```