```
package massim;

import java.util.Calendar;
import java.util.Random;

import massim.Team.TeamStepCode;

/**
 * The main class of the simulator.
 *
 * @author Omid Alemi
 * @version 1.2 2011/10/16
 *
 */
public class SimulationEngine {

    public static int[] colorRange = {1, 2, 3, 4, 5, 6};
    public static int[] actionCostsRange = {10, 40, 70, 100, 300, 400, 450,  500
};
    public static int numOfColors = colorRange.length;
    public static int numOfTeams;
    private int boardh = 10;
    private int boardw = 10;

    public static int disturbanceLevel;
    public static int mutualAwareness;

    private Team[] teams;
    Board mainBoard;
    int[][] actionCostsMatrix;
    RowCol[] goals;
    RowCol[] initAgentsPos;

    private int simCounter;
    private int[][] teamsScores;
    private int numOfRuns;

    private boolean debuggingInf = true;
    private boolean debuggingErr = true;

    public static enum SimRoundCode {
        SIMOK, SIMEND, SIMERR
    }

    /**
     * The constructor method
     *
     * @param teams                 The array of teams to be involved in
     *                              the simulations.
     */
    public SimulationEngine(Team[] teams) {
        logInf("SE created for " + teams.length + " teams.");
        this.teams = teams;
        SimulationEngine.numOfTeams = teams.length;
    }

    /**
     * Initializes the simulation engine for a new experiment. Each experiment
     * consists of a number of runs. The final scores of the experiment would be
```

```
     * the average of the scores over multiple runs
     *
     * @param numOfRuns             Number of desired runs for an identical
     *                              experiment setting.
     */
    public void initializeExperiment(int numOfRuns) {
        logInf("----- Experiment initialized for " + numOfRuns
                + " number of runs -----");
        teamsScores = new int[numOfTeams][numOfRuns];
        this.numOfRuns = numOfRuns;
    }

    /**
     * Initializes the simulation engine parameters for a new run. This includes
     * a new board setting, new action costs matrix, and possibly new positions
     * for initial agents' position and goals' position.
     *
     * The method also invokes the initializeRun() method of each team.
     */
    public void initializeRun() {
        logInf("--- The run initialized ---");
        simCounter = 0;
        mainBoard = Board.randomBoard(boardh, boardw);
        logInf("The board setting for this run is:\n" + mainBoard.toString());

        goals = new RowCol[Team.teamSize];
        for (int i = 0; i < Team.teamSize; i++)
            goals[i] = new RowCol(boardh - 1, boardw - 1);

        initAgentsPos = new RowCol[Team.teamSize];
        for (int i = 0; i < Team.teamSize; i++)
            initAgentsPos[i] = new RowCol(0, 0);

        Random rnd = new Random(Calendar.getInstance().getTimeInMillis());
        actionCostsMatrix = new int[Team.teamSize][numOfColors];
        for (int i = 0; i < Team.teamSize; i++)
            for (int j = 0; j < numOfColors; j++)
                actionCostsMatrix[i][j] = actionCostsRange[rnd
                        .nextInt(actionCostsRange.length)];

        for (int t = 0; t < numOfTeams; t++)
            teams[t].initializeRun(initAgentsPos, goals, actionCostsMatrix);
    }

    /**
     * Executes one round of the simulation.
     *
     * @return                  The proper simulation-round-code representing
     *                          the status of the round.
     */
    public SimRoundCode round() {
        simCounter++;
        logInf("Round #" + simCounter + " started ...");

        logInf("Chaning the board setting based on the disturbance level of "+
                disturbanceLevel);
        mainBoard.distrub(disturbanceLevel);

        TeamStepCode[] tsc = new TeamStepCode[teams.length];
```

```
        for (int t = 0; t < teams.length; t++) {
            tsc[t] = teams[t].round(mainBoard);
            logInf(teams[t].getClass().getSimpleName()
                    + " returned with the code: " + tsc[t].toString());
        }

        boolean allTeamsDone = true;
        for (int t = 0; t < teams.length; t++) {
            if (tsc[t] == TeamStepCode.OK) {
                allTeamsDone = false;
                break;
            }
        }

        if (allTeamsDone)
            return SimRoundCode.SIMEND;
        else
            return SimRoundCode.SIMOK;
    }

    /**
     * Executes the simulator for one whole run. This consists in invoking the
     * round() method of the engine until it indicates that it is either done
     * or there were a problem during the execution.
     *
     * @return              The final return code of the round method,
     *                      representing the return code of the run.
     */
    public SimRoundCode run() {
        logInf("-- The run started --");
        SimRoundCode src = SimRoundCode.SIMOK;
        while (src == SimRoundCode.SIMOK)
            src = round();
        logInf("-- The run ended --");
        return src;
    }

    /**
     * Executes the simulation for a whole experiment. The experiment consists
     * in multiple runs using the identical set of simulation parameters, but
     * with a new board and costs setting.
     *
     * @return              The average score of each team collected in
     *                      an array.
     */
    public int[] runExperiment() {
        logInf("---- The experiment started ----");
        for (int exp = 0; exp < numOfRuns; exp++) {
            initializeRun();
            run();
            for (int t = 0; t < numOfTeams; t++) {
                teamsScores[t][exp] = teams[t].teamRewardPoints();
                logInf("Team " + teams[t].getClass().getSimpleName()
                        + " scored " + teams[t].teamRewardPoints()
                        + " for this run.");
            }
        }
        logInf("---- The experiment ended ----");
```

```
        int[] averageTeamScores = new int[numOfTeams];
        for (int t = 0; t < numOfTeams; t++)
            averageTeamScores[t] = average(teamsScores[t]);

        return averageTeamScores;
    }

    /**
     * Calculates the average of the given integer array
     *
     * @param numbers       The array of integer numbers
     * @return              The average of the input array
     */
    private int average(int[] numbers) {
        int sum = 0;
        for (int i = 0; i < numbers.length; i++)
            sum += numbers[i];
        return sum / numbers.length;
    }

    /**
     * Prints the log message into the output if the information debugging level
     * is turned on (debuggingInf).
     *
     * @param msg            The desired message to be printed
     */
    private void logInf(String msg) {
        if (debuggingInf)
            System.out.println("[SimulationEngine]: " + msg);
    }

    /**
     * Prints the log message into the output if the error debugging level is
     * turned on (debuggingErr).
     *
     * @param msg             The desired message to be printed
     */
    private void logErr(String msg) {
        if (debuggingErr)
            System.err.println("[SimulationEngine]: " + msg);
    }
}
```

```
package massim;

import java.util.Random;

/**
 * Team.java
 *
 *
 * @author Omid Alemi
 * @version 1.2 2011/10/17
 */
public class Team {

    private static int nextID = 1; // for debugging purposes only
    private int id;

    public static int teamSize;
    public static int initResCoef;

    private CommMedium commMedium;
    private int[][] actionCostsMatrix;

    private static Random rnd1 = new Random();

    public static enum TeamStepCode {
        OK, DONE, ERR
    }

    private boolean debuggingInf = true;
    public int testRunCounter;

    /**
     * Default constructor
     */
    public Team() {
        id = nextID++;
        commMedium = new CommMedium();
    }

    /**
     * Called by the simulation engine to initialize the team and agents for
     * a new run. It should reset necessary variables.
     *
     * @param initAgentsPos                 Array of initial agents position
     * @param goals                         Array of initial goals position
     * @param actionCostMatrix              Matrix of action costs
     */
    public void initializeRun(RowCol[] initAgentsPos, RowCol[] goals,
            int[][] actionCostMatrix) {
        logInf("initilizing for a new run.");
        commMedium.clear();

        for (int i = 0; i < teamSize; i++)
            for (int j = 0; j < SimulationEngine.numOfColors; j++)
                this.actionCostsMatrix[i][j] = actionCostMatrix[i][j];
    }

    /**
     * Called by the simulation engine to start a new round of the simulation.
```

```
     *
     * @param board                         The current board representation
     * @return                              The proper TeamStepCode based on
     *                                      the team's current state after at
     *                                      the end of the round.
     */
    public TeamStepCode round(Board board) {
        logInf("starting a new round");
        for (int i = 0; i < Team.teamSize; i++) {

            int[][] probActionCostMatrix =
                new int[Team.teamSize][SimulationEngine.numOfColors];

            for (int p = 0; p < Team.teamSize; p++)
                for (int q = 0; q < SimulationEngine.numOfColors; q++)
                    if (rnd1.nextDouble() < SimulationEngine.mutualAwareness
                            || p == i)
                        probActionCostMatrix[p][q] =
                            actionCostsMatrix[p][q];
                    else
                        probActionCostMatrix[p][q] =
                            SimulationEngine.actionCostsRange[
                             rnd1.nextInt(SimulationEngine.actionCostsRange.leng
th)];

        }

        if (testRunCounter > 0) {  // For debugging purposes only;
            testRunCounter--;      // indicates when the team should be done
            return TeamStepCode.OK;
        } else {
            logInf(" is done!");
            return TeamStepCode.DONE;
        }
    }


    /**
     * To get the collective reward points of the team members
     *
     * @return                          The amount of reward points that all the
     *                                  team's agents own
     */
    public int teamRewardPoints() {
        int sum = 0;
        // for (Agent a: agents)
        // sum += a.rewardPoints();
        return sum;
    }

    /**
     * Prints the log message into the output if the information debugging level
     * is turned on (debuggingInf).
     *
     * @param msg                       The desired message to be printed
     */
    private void logInf(String msg) {
        if (debuggingInf)
            System.out.println("[Team " + id + "]: " + msg);
```

```
    }
}
```

```
package massim;

import java.util.Random;

/**
 * The class to hold the board settings
 *
 * @author Omid Alemi
 * @version 1.1
 */
public class Board {
    private static Random rndBoardGen = new Random();

    private int[][] mainBoard;

    private final int rows;
    private final int cols;

    /**
     * Constructor 1: just with the size
     *
     * @param r             The number of rows of the board
     * @param c             The number of columns of the board
     */
    public Board(int r, int c) {
        rows = r;
        cols = c;
        mainBoard = new int[rows][cols];
    }

    /**
     * Constructor 2: get the board setting and creating an exact copy
     *
     * @param board         The 2dim array, representing the board's
     *                      initial setting
     */
    public Board(Board board) {
        rows = board.rows();
        cols = board.cols();
        mainBoard = new int[rows][cols];

        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                this.mainBoard[i][j] = board.mainBoard[i][j];
    }

    /**
     * Returns the number of rows of the board
     *
     * @return              The number of rows of the board in int
     */
    public int rows() {
        return rows;
    }

    /**
     * Returns the number of columns of the board
     *
     * @return              The number of columns of the board in int
```

```
     */
    public int cols() {
        return cols;
    }

    /**
     * Sets the board setting to the giving setting
     *
     * @param initBoard     The input board setting to be the main board's
     *                      setting
     */
    public void setBoard(int[][] inputBoard) {

    }

    /**
     * Returns the board setting
     *
     * @return              2 dim array of int representing the board's
     *                      setting
     */
    public int[][] getBoard() {
        return mainBoard;
    }

    /**
     * Sets the value of one specific cell
     *
     * @param row           The row# of the desired cell
     * @param col           The column# of the desired cell
     * @param color         The new color for the desired cell
     */
    public void setCell(int row, int col, int color) {

    }

    /**
     * Static method; Creates a board with randomly filled values (colors).
     *
     * @return              The instance of the newly randomly generated board
     */
    public static Board randomBoard(int rows, int cols) {
        Board b = new Board(rows, cols);

        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                b.mainBoard[i][j] = rndBoardGen.nextInt(6);
        return b;
    }

    /**
     * Adds random values (disturbance) to the cells of the board. Each cell on
     * the board may be changed based on the probability defined by
     * disturbanecLevel
     *
     * @param disturbanceLevel      The level of disturbance, between 0 and 1.0
     */
    public void distrub(double disturbanceLevel) {
```

```
        Random rndColor = new Random();
        Random rndChange = new Random();

        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                if (rndChange.nextDouble() < disturbanceLevel)
                    mainBoard[i][j] = SimulationEngine.colorRange[rndColor
                            .nextInt(SimulationEngine.numOfColors)];
    }

    /**
     * Converts the current setting of the board into a string for debugging
     * purposes
     *
     * @return          The string representing the current setting of the board
     */
    @Override
    public String toString() {
        String out = "";

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++)
                out += mainBoard[i][j] + " ";
            out += "\n";
        }

        return out;
    }

    /**
     * Prints the costs associated with each square of the board based on the
     * given action costs set into a string Used for debugging purposes
     *
     * @param actionCosts       The action costs set of an agent
     * @return                  The string representation of the board;
     *                          displaying the costs of each cell
     */
    public String boardCostsToString(int actionCosts[]) {
        String out = "";
        int[] colorRange = SimulationEngine.colorRange;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int index = 0;
                for (int k = 0; k < colorRange.length; k++) {
                    int color = mainBoard[i][j];
                    if (color == colorRange[k])
                        index = k;
                }
                out += actionCosts[index] + "\t";
                ;

            }
            out += "\n";
        }

        return out;
    }
```

```
}
```

```
package massim;

import java.util.HashMap;

/**
 * CommMedium.java
 * Responsible for all the communications within a team of
 * agents
 *
 * @author Omid Alemi
 * @version 1.1 2011/10/07
 */
public class CommMedium {


    String[][] channels;
    int numOfChannels;
    /**
     * The default constructor
     *
     */
    public CommMedium() {

        numOfChannels = Team.teamSize;
        // Initializing all the channels
        channels = new String[numOfChannels][numOfChannels];
        for (int i=0;i<numOfChannels;i++)
            for (int j=0;j<numOfChannels;j++)
                channels[i][j]="";
    }

    /**
     * Puts the msg into the receiver's special channel for the sender
     *
     * @param sender                The sender agent's id
     * @param receiver              The receiver agent's id
     * @param msg                   The message
     */
    public void send(int sender, int receiver, String msg) {

        if (receiver != sender)
            channels[sender][receiver] = msg;
    }

    /**
     * Puts the msg into all the agent's special channels for the sender
     *
     * @param sender                The sender agent's id
     * @param msg                   The message
     */
    public void broadcast(int sender, String msg) {

        for (int i=0;i<Team.teamSize;i++)
            if (i!=sender)
                channels[sender][i] = msg;
    }


    /**
```

```
     * Returns the next available message in the receiver's incoming communicati
on channels.
     * Returns an empty message if there is no message left on the channels
     *
     * @param receiver              The id of the receiver agent
     * @return                      The message
     */
    public String receive(int receiver) {

        String out="";

        for(int i=0;i<channels.length;i++)
        {
            if (!channels[i][receiver].isEmpty())
            {
                out = channels[i][receiver];
                channels[i][receiver] = "";
                return out;
            }
        }
        return out;
    }

    /**
     * To check whether the communication medium is empty. Means there
     * were no communication during the last iteration
     *
     * @return                  true if all the channels for all the agents are
empty.
     *                          false otherwise
     */
    public boolean isEmpty() {

        for (int i=0;i<Team.teamSize;i++)
            for (int j=0;j<Team.teamSize;j++)
                if (channels[i][j] != "")
                    return false;
        return true;
    }

    /**
     * Clears all the channels
     */
    public void clear() {
        for (int i=0;i<numOfChannels;i++)
            for (int j=0;j<numOfChannels;j++)
                channels[i][j]="";
    }

    /**
     * Used for the debugging purposes
     * Generates a string representation of all the communication channels
     * and their values
     */
    @Override
    public String toString() {
        String s = "";
        for (int i=0;i<channels[0].length;i++)
        {
```

```
            s += "[Agent "+ i+ "'s incoming channels: ]\n";

            for (int j=0;j<channels.length;j++)
            {
                if (i!=j)
                {
                    s += "Agent "+ j+ ":";
                    s += channels[j][i];
                    s += "\n";
                }
            }
        }
        return s;
    }

}
```

```
package massim.agents.dummy;

import java.util.Random;

import massim.RowCol;
import massim.Team;

public class DummyTeam extends Team {

    /**
     * The default constructor
     */
    public DummyTeam() {
        super();

    }

    /**
     * The overridden initializeRun method of the Team class.
     *
     * This calls the same method of the superclass first.
     *
     */
    @Override
    public void initializeRun(RowCol[] initAgentsPos, RowCol[] goals, int[][]act
ionCostMatrix) {
        super.initializeRun(initAgentsPos, goals, actionCostMatrix);
        testRunCounter = 10 + (new Random()).nextInt(5);
    }


    /**
     * For debugging purposes only:
     *
     * The overridden teamRewardPoints method to return a dummy amount
     * of reward points.
     *
     * @return                  The amount of reward points.
     */
    @Override
    public int teamRewardPoints()
    {
        Random rnd = new Random();
        return rnd.nextInt(10000);
    }

}
```

```
package massim.agents.dummy;

import java.util.Random;

import massim.RowCol;
import massim.Team;

public class UselessTeam extends Team{


    /**
     * The default constructor
     */
    public UselessTeam() {
        super();
    }

    /**
     * The overridden initializeRun method of the Team class.
     *
     * This calls the same method of the superclass first.
     *
     */
    @Override
    public void initializeRun(RowCol[] initAgentsPos, RowCol[] goals, int[][]act
ionCostMatrix) {
        super.initializeRun(initAgentsPos, goals, actionCostMatrix);
        testRunCounter = 10 + (new Random()).nextInt(5);
    }

    /**
     * For debugging purposes only:
     * private void logInf(String msg) {
        if (debuggingInf)
            System.out.println("[Useless Team]: " + msg);
    }
     * The overridden teamRewardPoints method to return a dummy amount
     * of reward points.
     *
     * @return                  The amount of reward points.
     */
    @Override
    public int teamRewardPoints()
    {
        Random rnd = new Random();
        return rnd.nextInt(10000);
    }
}
```

Table of Contents