```
package massim;

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

import massim.Team.TeamRoundCode;

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

    public static int[] colorRange = {0, 1, 2, 3, 4, 5};
    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 double disturbanceLevel;

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

    private int roundCounter;
    private int[][] teamsScores;
    private int numOfRuns;

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

    /**
     * SIMOK: The round executed without any problem and there is
     *        at least one active team.
     *
     * SIMEND: All the teams are done.
     *
     * SIMERR: There was a problem in the current round.
     */
    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 in a number of runs.
     * The final score of each team for each run will be stored in an
     * array.
     *
     * @param numOfRuns             Number of desired runs for current
     *                              experiment setting.
     */
    public void initializeExperiment(int numOfRuns) {
        logInf("----- Experiment initialized for " + numOfRuns
                + " number of runs -----");
        teamsScores = new int[numOfTeams][numOfRuns];
        this.numOfRuns = numOfRuns;
    }

    /**
     * Prepares 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 Team.initializeRun() for all teams.
     */
    public void initializeRun() {
        logInf("--- The run initialized ---");
        roundCounter = 0;
        mainBoard = Board.randomBoard
                    (boardh, boardw,SimulationEngine.colorRange);

        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] = randomPos(boardh, boardw);

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

        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.
     *
```

```
     * Each round of the simulation consist in updating the board; executing
     * each team; and checking the current status of the simulation.
     *
     * It is possible to implement error handling mechanisms for this method.
     *
     * @return              The proper simulation-round-code representing
     *                      the status of the round.
     */
    public SimRoundCode round() {
        roundCounter++;
        logInf("Round #" + roundCounter + " started ...");

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

        TeamRoundCode[] tsc = new TeamRoundCode[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] == TeamRoundCode.OK) {
                allTeamsDone = false;
                break;
            }
        }

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

    /**
     * Executes the simulator for one whole run.
     *
     * A run consists in invoking the round() until it indicates that it is
     * either done or there was a problem during the execution.
     *
     * @return              The return code of the last round method
     *                      invocation, 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 one whole experiment.
     *
```

```
     * A experiment consists in multiple runs using the identical set
     * of parameters, but with a new board and costs settings.
     *
     * @return              The score of each team averaged over multiple
     *                      runs.
     */
    public int[] runExperiment() {
        logInf("---- The experiment started ----");
        for (int r = 0; r < numOfRuns; r++) {
            initializeRun();
            run();
            for (int t = 0; t < numOfTeams; t++) {
                teamsScores[t][r] = 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.
     *
     * Note: it calculates the average using a double division then
     * rounding the result to the nearest integer.
     *
     * @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 (int) Math.round((double)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);
    }

    /**
     * Generates a random position within the specified range
     *
     * @param h                 The height of the board
     * @param w                 The width of the board
     * @return                  The generated position
     */
    private RowCol randomPos(int h, int w) {
        Random rnd = new Random();

        return new RowCol(rnd.nextInt(h),rnd.nextInt(w));
    }
}
```

```
package massim;

import java.util.Random;

import massim.Agent.AgGameStatCode;
import massim.Agent.AgCommStatCode;

/**
 * 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;
    public static double mutualAwareness;

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

    AgGameStatCode[] agentsGameStatus = new AgGameStatCode[Team.teamSize];
    AgCommStatCode[] agentsCommStatus = new AgCommStatCode[Team.teamSize];
    private static Random rnd1 = new Random();


    /**
     * OK: The round executed without any problem and there is
     *       at least one active agent.
     *
     * DONE: All the agents are done.
     *
     * ERR: There was a problem in the current round.
     */
    public static enum TeamRoundCode {
        OK, DONE, ERR
    }

    private boolean debuggingInf = true;
    public int testRunCounter;

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

        actionCostsMatrix =
            new int[Team.teamSize][SimulationEngine.numOfColors];
    }

    /**
```

```
     * Initializes the team and agents for a new run.
     *
     * Called by the simulation engine (SimulationEngine.initializeRun())
     * It should reset necessary variables values.
     *
     * @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];

        for (int i = 0; i < teamSize; i++)
            agentsGameStatus[i] = AgGameStatCode.READY;
    }

    /**
     * Starts a new round of the simulation for this team.
     *
     * Called by the simulation engine (SimulationEngine.round()).
     *
     * It is possible to implement error handling mechanisms for this method.
     *
     * @param board                         The current board representation
     * @return                              The proper TeamRoundCode based on
     *                                      the team's current state.
     */
    public TeamRoundCode round(Board board) {
        logInf("********");
        logInf("starting a new round");

        /* Initialize round for agents */
        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() < Team.mutualAwareness
                            || p == i)
                        probActionCostMatrix[p][q] =
                            actionCostsMatrix[p][q];
                    else
                        probActionCostMatrix[p][q] =
                            SimulationEngine.actionCostsRange[
                             rnd1.nextInt(
                                    SimulationEngine.actionCostsRange.length)];

            if (agentsGameStatus[i] != AgGameStatCode.BLOCKED)
                agents[i].initializeRound(board, probActionCostMatrix);

            agentsCommStatus[i] = AgCommStatCode.NEEDING_TO_SEND;
        }
```

```
        /* Communication Cycles */
        boolean allDoneComm = false;
        logInf("");
        while(!allDoneComm) {

            for (int i = 0; i < Team.teamSize; i++)
            {
                /* TODO: Double check the need of first condition */
                if (agentsGameStatus[i] != AgGameStatCode.BLOCKED &&
                        agentsCommStatus[i] != AgCommStatCode.DONE)
                    agents[i].sendCycle();
            }
            allDoneComm = true;

            for (int i = 0; i < Team.teamSize; i++)
            {
                /* TODO: Double check the need of first condition */
                if (agentsGameStatus[i] != AgGameStatCode.BLOCKED &&
                        agentsCommStatus[i] != AgCommStatCode.DONE)
                    agentsCommStatus[i] = agents[i].receiveCycle();

                if (agentsGameStatus[i] != AgGameStatCode.BLOCKED &&
                    agentsCommStatus[i] != AgCommStatCode.DONE)
                    allDoneComm = false;
            }
        }

        /* Finalize the round for agents */

        boolean allDone = true;
        for (int i = 0; i < Team.teamSize; i++)
        {

        /* If the agent were blocked before, don't call it as it doesn't have
           enough resources to help itself nor its teammates.
           However, call those who has reached the goal, they may help others.
        */

            if (agentsGameStatus[i] != AgGameStatCode.BLOCKED)
                agentsGameStatus[i]  = agents[i].finalizeRound();

            if (agentsGameStatus[i] != AgGameStatCode.REACHED_GOAL &&
                    agentsGameStatus[i] != AgGameStatCode.BLOCKED)
                allDone = false;
        }

        commMedium.clear();

        if (allDone)
            return TeamRoundCode.DONE;
        else
            return TeamRoundCode.OK;
    }


    /**
     * To get the collective reward points of the team members
     *
```

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

    /**
     * Enables access to the specified agent.
     *
     * @param id                        The id of the agent
     * @return                          The instance of the agent
     */
    protected Agent agent(int id) {
        return agents[id];
    }

    /**
     * Sets the agents of the team.
     *
     *
     * @param agents                    The array of agents.
     */
    protected void setAgents(Agent[] agents) {
        this.agents = agents;
    }
    /**
     * 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.agents.dummy;

import massim.RowCol;
import massim.Team;

/**
 *
 *
 * @author Omid Alemi
 * @version 2.0 2011/10/31
 */
public class DummyTeam extends Team {

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

        DummyAgent[] dummyAgents = new DummyAgent[Team.teamSize];

        for(int i=0;i<Team.teamSize;i++)
            dummyAgents[i] = new DummyAgent(i);

        setAgents(dummyAgents);
    }

    /**
     * The overridden Team.initializeRun() method.
     *
     * This calls the same method of the superclass first.
     *
     * Initialized the agents, giving them their initial position, goal
     * position, action costs, and their initial resources based on their
     * path length.
     */
    @Override
    public void initializeRun(
        RowCol[] initAgentsPos, RowCol[] goals, int[][]actionCostsMatrix) {

        super.initializeRun(initAgentsPos, goals, actionCostsMatrix);

        for(int i=0;i<Team.teamSize;i++)
        {
            int pathLength = calcDistance(initAgentsPos[i], goals[i]);

            agent(i).initializeRun(initAgentsPos[i], goals[i],
                    actionCostsMatrix[i], pathLength * initResCoef);
        }
    }

    /**
     * Calculates the distance between two points in a board.
     *
     * @param start                 The position of the starting point
     * @param end                   The position of the ending point
     * @return                      The distance
     */
    private int calcDistance(RowCol start, RowCol end) {
```

```
        return  Math.abs(end.row-start.row) + Math.abs(end.col-start.col) + 1;
    }

    /**
     * For debugging purposes only:
     *
     * The overridden Team.teamRewardPoints() method to return a dummy amount
     * of reward points.
     *
     * @return                  The amount of reward points.
     */
    @Override
    public int teamRewardPoints()
    {
        int sum = 0;

        for(int i=0;i<Team.teamSize;i++)
            sum += agent(i).rewardPoints();

        return sum;
    }

}
```

```
package massim.agents.dummy;

import java.util.Random;

import massim.Agent;
import massim.Board;
import massim.RowCol;

public class DummyAgent extends Agent {

    private boolean debuggingInf = true;

    private int procrastinateCount;
    private int procrastinateLevel;


    enum DummyStates {S_INIT, S_PROC, R_PROC, R_MOVE, R_BLOCKED};

    DummyStates state;

    /**
     * The constructor
     *
     * @param id            The given id of the agent
     */
    public DummyAgent(int id) {
        super(id);
    }


    /**
     * Initializes the agent for a new run.
     *
     * Called by Team.initializeRun()

     *
     * @param initialPosition           The initial position of this agent
     * @param goalPosition              The goal position for this agent
     * @param actionCosts               The agent's action costs vector
     * @param initResourcePoints        The initial resource points given
     *                                  to the agent by its team.
     */
    @Override
    public void initializeRun(RowCol initialPosition, RowCol goalPosition,
            int[] actionCosts, int initResourcePoints) {

        super.initializeRun(initialPosition, goalPosition,
                actionCosts,initResourcePoints);

        logInf("Initialized for a new run.");
        logInf("My initial resource points = "+resourcePoints());
        logInf("My goal position: " + goalPos().toString());

    }

    /**
     * Initializes the agent for a new round of the game.
     *
     *
```

```
     * @param board                    The game board
     * @param actionCostsMatrix        The matrix containing the action costs
     *                                 for all the agents in the team (depends
     *                                 on the level of mutual awareness in the
     *                                 team)
     */
    @Override
    protected void initializeRound(Board board, int[][] actionCostsMatrix) {
        super.initializeRound(board, actionCostsMatrix);

        logInf("Starting a new round ...");

        logInf("My current position: " + pos().toString());
        if (path() == null)
        {
            findPath();
            logInf("Chose this path: "+ path().toString());
        }

        state = DummyStates.S_INIT;
        logInf("Set the inital state to +"+state.toString());

        setRoundAction(actionType.SKIP);

        procrastinateLevel = (new Random()).nextInt(4);
        procrastinateCount = 0;
    }

    /**
     * A dummy send cycle method.
     *
     * Will alternate between send and receive states.
     *
     */
    @Override
    protected AgCommStatCode sendCycle() {
        AgCommStatCode returnCode = AgCommStatCode.DONE;
        logInf("Send Cycle");

        switch (state) {
        case S_INIT:
            RowCol nextCell = path().getNextPoint(pos());
            int cost = getCellCost(nextCell);
            if (cost  <= resourcePoints())
                setState(DummyStates.R_PROC);
            else
                setState(DummyStates.R_BLOCKED);
            returnCode = AgCommStatCode.NEEDING_TO_REC;
            break;
        case S_PROC:
            procrastinateCount++;
            if (procrastinateCount > procrastinateLevel)
                setState(DummyStates.R_MOVE);
            else
                setState(DummyStates.R_PROC);
            returnCode = AgCommStatCode.NEEDING_TO_REC;
            break;
        default:
            logErr("Undefined state: " + state.toString());
```

```
        }

        return returnCode;
    }

    /**
     * A dummy receive cycle method.
     *
     * Will alternate between send and receive cycles for 3 times.
     * Then will transit to a final state.
     *
     */
    @Override
    protected AgCommStatCode receiveCycle() {
        AgCommStatCode returnCode = AgCommStatCode.DONE;

        logInf("Receive Cycle");

        switch (state) {
        case R_PROC:
            setState(DummyStates.S_PROC);
            returnCode = AgCommStatCode.NEEDING_TO_SEND;
            break;
        case R_MOVE:
            logInf("Setting current action to do my own move");
            setRoundAction(actionType.OWN);
            returnCode = AgCommStatCode.DONE;
            break;
        case R_BLOCKED:
            setRoundAction(actionType.FORFEIT);
            returnCode = AgCommStatCode.DONE;
            break;
        default:
            logErr("Undefined state: " + state.toString());
        }

        return returnCode;
    }

    /**
     * Finalizes the round by moving the agent.
     *
     * Also determines the current state of the agent which can be
     * reached the goal, blocked, or ready for next round.
     *
     * @return                          Returns the current state
     */
    @Override
    protected AgGameStatCode finalizeRound() {

        logInf("Finalizing the round ...");

        if (pos().equals(goalPos()))
        {
            logInf("Reached the goal");
            return AgGameStatCode.REACHED_GOAL;
        }
        else
        {
```

```
            if (act())
                return AgGameStatCode.READY;
            else  /*TODO: The logic here should be changed!*/
                {
                    logInf("Blocked!");
                    return AgGameStatCode.BLOCKED;
                }
        }
    }


    /**
     * 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("[DummyAgent " + id() + "]: " + msg);
    }

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

    /**
     * Changes the current state of the agents state machine.
     *
     * @param newState              The new state
     */
    private void setState(DummyStates newState) {
        logInf("In "+ state.toString() +" state");
        state = newState;
        logInf("Set the state to +"+state.toString());
    }

    /**
     * Agent's move action.
     *
     * Moves the agent to the next position if possible
     *
     * TODO: Needs to be extended to perform help.
     *
     * @return
     */
    private boolean move() {

        RowCol nextCell = path().getNextPoint(pos());
        if (pos().equals(nextCell))
        {
```

```
            logInf("Can not move from "+pos() +" to itself!");
            return false;
        }
        else
        {
            logInf("Moved from "+pos() +" to "+ nextCell);

            int cost = getCellCost(nextCell);
            decResourcePoints(cost);
            setPos(nextCell);
            return true;
        }
    }

    /**
     * The DummyAgent performs its own action (move) here.
     *
     * @return                  The same as what move() returns.
     */
    @Override
    protected boolean doOwnAction() {
        return move();
    }

}
```

```
package tests;

import java.util.Scanner;

import massim.Agent;
import massim.SimulationEngine;
import massim.Team;
import massim.agents.dummy.DummyTeam;
import massim.agents.dummy.UselessTeam;

/**
 * Simulation Engine/Teams interaction test.
 *
 * @author Omid Alemi
 * @version 2011/10/17
 */
public class TeamAgentTester {

    public static void main(String[] args)
    {

        multipleExperiments();
    }


    /**
     * To demonstrate how to use the SimulatinEngine to perform
     * multiple experiments by changing some parameters.
     */
    public static void multipleExperiments()
    {
        int numberOfRuns = 4;

        /* Create the teams involved in the simulation */
        Team.teamSize = 4;
        Team[] teams = new Team[1];
        teams[0] = new DummyTeam();


        /* Create the SimulationEngine */
        SimulationEngine se = new SimulationEngine(teams);

        /* The experiments loop */
        for (int exp=0;exp<11;exp++)
        {
            /* Set the experiment-wide parameters: */
            /* teams-wide, SimulationEngine, etc params */

            Team.initResCoef = 80;
            Agent.cellReward = 50;

            /* vary the disturbance: */
            SimulationEngine.disturbanceLevel = 0.1 * exp;

            /* Initialize and run the experiment */
            se.initializeExperiment(numberOfRuns);
            int[] teamScores = se.runExperiment();

            /* Print the results */
```

```
            for (int i=0;i<teams.length;i++)
                System.out.println("Exp"+exp+": disturbance level="+
                        SimulationEngine.disturbanceLevel+"; "+
                        teams[i].getClass().getSimpleName()+
                        " average score = "+teamScores[i]);
            (new Scanner(System.in)).nextLine();

        }

    }


}
```

```
package massim;

/**
 * Agent.java An abstract class for all the agents to be used in the simulator
 *
 * @author Omid Alemi
 * @version 2.0 2011/10/31
 */
public abstract class Agent {

    public static int cellReward;

    protected static enum AgGameStatCode {
        READY, REACHED_GOAL, RESOURCE_BLOCKED, BLOCKED
    }

    protected static enum AgCommStatCode {
        DONE, NEEDING_TO_SEND, NEEDING_TO_REC
    }

    protected static enum actionType {
        OWN, HELP_ANOTHER, HAS_HELP, SKIP, FORFEIT
    }

    private int id;

    private int[] actionCosts;
    private Path path;

    private int resourcePoints = 0;

    private RowCol pos;
    private RowCol goalPos;
    private Board theBoard;

    private actionType thisRoundAction = actionType.SKIP;

    /**
     * The constructor.
     *
     * The team will pass the id to the agent.
     *
     * @param id            The id of the agent being created.
     */
    public Agent(int id) {
        this.id = id;
        goalPos = null;
        pos = null;
        theBoard = null;
        path = null;
    }

    /**
     * Initializes the agent for a new run.
     *
     * Called by Team.initializeRun()

     *
     * @param initialPosition           The initial position of this agent
```

```
     * @param goalPosition              The goal position for this agent
     * @param actionCosts               The agent's action costs vector
     * @param initResourcePoints        The initial resource points given
     *                                  to the agent by its team.
     */
    public void initializeRun(RowCol initialPosition, RowCol goalPosition,
            int[] actionCosts, int initResourcePoints) {

        goalPos = null;
        pos = null;
        theBoard = null;
        path = null;

        this.pos = initialPosition;
        this.goalPos = goalPosition;

        this.actionCosts = new int[actionCosts.length];
        System.arraycopy(actionCosts, 0, this.actionCosts, 0,
                actionCosts.length);

        incResourcePoints(initResourcePoints);
    }

    /**
     * Initializes the agent for a new round of the game.
     *
     *
     * @param board                     The game board
     * @param actionCostsMatrix         The matrix containing the action costs
     *                                  for all the agents in the team (depends
     *                                  on the level of mutual awareness in the
     *                                  team)
     */
    protected void initializeRound(Board board, int[][] actionCostsMatrix) {
        this.theBoard = board;

    }

    /**
     * Enables the agent to send its outgoing messages (if any)
     *
     *
     * @return                          The current state of the agent
     */
    protected AgCommStatCode sendCycle() {

        return AgCommStatCode.DONE;
    }

    /**
     * Enables the agent to receive its incoming messages (if any)
     *
     *
     * @return                          The current state of the agent
     */
    protected AgCommStatCode receiveCycle() {


        return AgCommStatCode.DONE;
```

```
    }

    /**
     * Enables the agent to perform any actions for this round of
     * the game.
     *
     * @return                          The status of the agent after
     *                                  this round
     */
    protected AgGameStatCode finalizeRound() {

        return AgGameStatCode.READY;
    }

    /**
     * Enables the agent to get their id
     *
     * @return                          The id of the agent
     */
    protected int id() {
        return id;
    }

    /**
     * Returns the amount reward points that the agent has earned.
     *
     * @return                          The reward points
     */
    public int rewardPoints() {

        return path.getIndexOf(pos) * cellReward;
    }

    /**
     * Returns the amount of resource points that the agent has earned.
     *
     * @return                          The resource points
     */
    protected int resourcePoints() {
        return resourcePoints;
    }

    /**
     * Increases the resource points by the specified amount.
     *
     * @param amount                    The amounts to be added
     */
    public void incResourcePoints(int amount) {
        resourcePoints += amount;
    }

    /**
     * Decreases the resource points by the specified amount.
     *
     * @param amount                    The amounts to be subtracted
     */
    protected void decResourcePoints(int amount) {
        if (resourcePoints - amount < 0)
            System.err.println("ERROR: decreasing too much resource points!");
```

```
        else
            resourcePoints -= amount;
    }

    /**
     * Enables the agent to access its current position.
     *
     * @return                          The current position
     */
    protected RowCol pos() {
        return pos;
    }

    /**
     * Sets the position of the agent
     *
     * @param newPos                    The new position
     */
    protected void setPos(RowCol newPos) {
        pos = newPos;
    }

    /**
     * Enables the agent to access its goal position.
     *
     * @return                          The position of the goal
     */
    protected RowCol goalPos() {
        return goalPos;
    }

    /**
     * Enables the agent to access its action costs vector.
     *
     * @return                          The action costs vector
     *                                  of the agent
     */
    protected int[] actionCosts() {

        return actionCosts;
    }

    /**
     * Returns the cost of a given cell for this agent
     *
     * @param cell                      The position of the cell
     * @return                          The cost associated with
     *                                  the color of the given
     *                                  cell
     */
    protected int getCellCost(RowCol cell) {
        int color = theBoard.getBoard()[cell.row][cell.col];
        return actionCosts()[color];
    }

    /**
     * Returns the cost of a given cell based on the given actions
     * cost vector.
     *
```

```
     * @param cell                      The position of the cell
     * @param actCost                   The action costs vector
     * @return                          The cost associated with
     *                                  the color of the given
     *                                  cell
     */
    protected int getCellCost(RowCol cell, int[] actCost) {

        int color = theBoard.getBoard()[cell.row][cell.col];
        return actCost[color];
    }

    /**
     * Enables the agent to access to the game board
     *
     * @return                          The instance of the board
     */
    protected Board theBoard() {
        return theBoard;
    }

    /**
     * Finds the lowest cost path among shortest paths of a rectangular board
     * based on the Polajnar's algorithm V2.
     *
     * The method uses the agents position as the starting point and the goal
     * position as the ending point of the path.
     */
    protected void findPath() {
        PolajnarPath2 pp = new PolajnarPath2();
        Path shortestPath = new Path(pp.findShortestPath(
                boardToCosts(theBoard.getBoard(), actionCosts), pos, goalPos));
        path = new Path(shortestPath);
    }

    /**
     * Creates a two dimensional array representing the cell cost
     * based on the given action costs vector.
     *
     * This method is used by the path finding algorithm.
     *
     * @param board                     The game board setting
     * @param actionCosts               The action costs
     * @return                          The 2dim array of costs
     */
    private int[][] boardToCosts(int[][] board, int[] actionCosts) {
        int[][] costs = new int[board.length][board[0].length];

        for (int i = 0; i < costs.length; i++)
            for (int j = 0; j < costs[0].length; j++)
                costs[i][j] = actionCosts[board[i][j]];

        return costs;
    }

    /**
     * Enables the agent to access its path
     *
     * @return                          The instance of the path.
```

```
     */
    protected Path path() {
        return path;
    }



    /**
     * Sets the type of the action that is going to be performed
     * in this round.
     *
     * @param a                     The action type
     */
    protected void setRoundAction(actionType a) {
        thisRoundAction = a;
    }

    /**
     * Enables the agent to get the action type for current round.
     *
     * @return                      The action type
     */
    protected actionType getRoundAction() {
        return thisRoundAction;
    }

    /**
     * Enables the agent to perform its own action.
     *
     * To be overriden by the agent if necessary.
     *
     * @return                      true if successful/false o.w.
     */
    protected boolean doOwnAction() {

        return true;
    }

    /**
     * Enables the agent to perform an action on behalf of another
     * agent (Help).
     *
     * To be overriden by the agent if necessary.
     *
     * @return                      true if successful/false o.w.
     */
    protected boolean doHelpAnother() {

        return true;
    }

    /**
     * Enables the agent do any bookkeeping while receiving help.
     *
     * To be overriden by the agent if necessary.
     *
     * @return                      true if successful/false o.w.
     */
    protected boolean doGetHelpAction() {
```

```
        return true;
    }


    protected boolean act() {

        boolean result = false;

        switch (thisRoundAction) {
        case OWN:
            result = doOwnAction();
            break;
        case HAS_HELP:
            result = doGetHelpAction();
            break;
        case HELP_ANOTHER:
            result = doHelpAnother();
            break;
        case SKIP:
            result = true;
            break;
        case FORFEIT:
            result = false;
            break;
        }

        return result;
    }
}
```

Table of Contents