

```

1 package ch.ethz.icr.tutorial.experipd;
2
3 import uchicago.src.sim.analysis.LocalDataRecorder;
4 import uchicago.src.sim.engine.SimInit;
5
6 /**
7  * Evolutionary game based on grid with batch capacity
8  * <p/>
9  * This is the new class that had been added to ExperIPD in order to make
10 * it possible to do batch runs. The parameter file params.txt will be read
11 * upon starting the simulation and a data file data.txt will be generated.
12 * The latter file contains the number of actors of each type at the end of
13 * the preset number of iterations (@see stoppingTime).
14 * We create the Batch class through subclassing from the Model class just
15 * like in the case of ModelGUI.
16 * <p/>
17 * This is the fifth step of the tutorial. This stage in the tutorial sequence
18 * uses the spatial grid from GridIPD. The main novelty is the ModelBatch class
19 * that provides batch access to the model. Thus, all files are identical to
20 * GridIPD except for ModelBatch
21 * <p/>
22 * NOTE: The program should now be started by running either ModelGUI or
23 * ModelBatch. The choice between these is up to the user. However, selecting
24 * ModelBatch suppresses the GUI, and reads the parameters from a file and
25 * records the results to another file.
26 * <p/>
27 * For the original model, see Cohen, Riolo, and Axelrod:
28 * The Emergence of Social Organization in the Prisoner's Dilemma
29 * (SFI Working Paper), 1999. http://www.santafe.edu/ (follow publications link).
30 *
31 * @author Luc Girardin
32 * @author Lars-Erik Cederman
33 * @author Laszlo Gulyas
34 * @version 2.0
35 */
36 public final class ModelBatch extends Model {
37
38     // Batch variables
39     private int numOfTimeSteps; // Defines the length of each batch replication run.
40     private LocalDataRecorder recorder; // Repast's mechanism for data recording.
41
42     /**
43      * The batch module's default variable settings have to be defined.
44      * Note that all the other default settings are defined in Model.setup().
45      * Also note that all settings here can be overridden by the parameter file.
46      */
47     public final void setup() {
48         // Initializing the original model first
49         super.setup();
50
51         // Specify the parameters to be manipulated by RePast's parameter
52         // mechanism (i.e. set from the parameter file).
53         params = new String[]{"Topology", "Neighborhood",
54                               "NumOfTimeSteps", "RngSeed", "WorldSize",
55                               "PALLC", "PTFT", "PATFT", "PALLD", "PAdapt"};
56
57         // Initializing the batch part
58         numOfTimeSteps = 200; // This sets the number of iterations.
59     }
60
61     /**
62      * The method to build the Model's internals
63      * Here we build the model instrumentation after the model itself has

```

```
64   * been built. We make sure that the data recorder is initialized.
65   */
66   public final void buildModel() {
67       // Set the number of simulation steps
68       setStoppingTime(numOfTimeSteps);
69
70       // Build the Batch-part
71       // Create the DataRecorder object and specify the name of the output file.
72       // (For details see Repast's appropriate "How to" document.)
73       recorder = new LocalDataRecorder("data.txt", this);
74
75       // Now we add four columns in our output file. The strings that appear as
76       // the last arguments refer to method names that appear below.
77       recorder.createNumericDataSource("ALLC", this, "computeAllC");
78       recorder.createNumericDataSource("TFT", this, "computeTFT");
79       recorder.createNumericDataSource("ATFT", this, "computeATFT");
80       recorder.createNumericDataSource("ALLD", this, "computeAllD");
81
82       recorder.createNumericDataSource("Payoff", this, "computePayoff", 1, 4);
83       recorder.setDelimiter(" ");
84
85       // Build the original model last
86       super.buildModel();
87
88   }
89
90   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
91   // Iterated methods ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
92   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
93
94   /**
95    * We suppress the original model's reportResults() method by overriding it.
96    * We do so, in order to prevent excessive output on the screen.
97    */
98   public void reportResults() {
99   }
100
101   /**
102    * We override the SimpleModel's atEnd() method to add functionality that
103    * writes the simulation results to a file at the end of each replication run.
104    */
105   public final void atEnd() {
106       super.atEnd();
107
108       // We need to calculate the results first. For this, we use
109       // the original model's reportResult() method.
110       // Note that by explicitly calling super.reportResults() we
111       // circumvent the 'suppression' above.
112       super.reportResults();
113
114       // Record the results into the DataRecorder...
115       recorder.record();
116
117       // ... and write it into the file right away.
118       recorder.write();
119   }
120
121   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
122   // RePast Parameter Panel Methods ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
123   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
124
125   // In the following we provide get/set methods for all the parameters
126   // we listed in params (see the setup() method)
```

```
127 // Note that except for the two first methods, these declarations are
128 // identical to the access methods in ModelGUI.
129
130 public final int getNumOfTimeSteps() {
131     return numOfTimeSteps;
132 }
133
134 public final void setNumOfTimeSteps(final int v) {
135     numOfTimeSteps = v;
136 }
137
138 public final int getWorldSize() {
139     return worldSize;
140 }
141
142 public final void setWorldSize(final int n) {
143     worldSize = n;
144 }
145
146 public final double getPALLC() {
147     return pALLC;
148 }
149
150 public final void setPALLC(final double p) {
151     pALLC = p;
152 }
153
154 public final double getPTFT() {
155     return pTFT;
156 }
157
158 public final void setPTFT(final double p) {
159     pTFT = p;
160 }
161
162 public final double getPATFT() {
163     return pATFT;
164 }
165
166 public final void setPATFT(final double p) {
167     pATFT = p;
168 }
169
170 public final double getPALLD() {
171     return pALLD;
172 }
173
174 public final void setPALLD(final double p) {
175     pALLD = p;
176 }
177
178 public final double getPAdapt() {
179     return pAdapt;
180 }
181
182 public final void setPAdapt(final double p) {
183     pAdapt = p;
184 }
185
186 public int getTopology() {
187     return topology;
188 }
189
```

```
190     public void setTopology(int topology) {
191         this.topology = topology;
192     }
193
194     public int getNeighborhood() {
195         return neighborhood;
196     }
197
198     public void setNeighborhood(int neighborhood) {
199         this.neighborhood = neighborhood;
200     }
201
202     public final double computeAllC() {
203         return (double) num[ALLC];
204     }
205
206     public final double computeTFT() {
207         return (double) num[TFT];
208     }
209
210     public final double computeATFT() {
211         return (double) num[ATFT];
212     }
213
214     public final double computeAllD() {
215         return (double) num[ALLD];
216     }
217
218     public final double computePayoff() {
219         return averagePayoff;
220     }
221
222     //////////////////////////////////////
223     // Creating and starting your model //////////////////////////////////////
224     //////////////////////////////////////
225     public static void main(final String[] args) {
226         final SimInit init = new SimInit();
227         // We MUST create a ModelBatch object instead of an instance of Model
228         // or that of a ModelGUI, in order to get the Batch functionality.
229         final Model m = new ModelBatch();
230         // Note the differences between the parameters when loading a batch-mode
231         // model (compared to that of a GUI-mode one, @see ModelGUI).
232         // The second parameter specifies the name of the parameter file, while
233         // the third one declares that the model is to be run in batch-mode.
234         init.loadModel(m, "params.txt", true);
235     }
236 }
```