0.1 Game of Life: Repast Java implementation

0.2 Michelle Gosse

0.3 Using two agent classes

0.3.1 Eclipse setup for Java

Open Repast. On the top right-hand side, make sure that the **Java perspective** is selected.

The project uses classes imported from Repast. Instead of manually importing each time a new class is called, Eclipse can be setup to automatically import the relevant class. The import statements will be automatically positioned in the class you are creating, in a set immediately following the package statement in your class.

From the menu bar, select Window, Preferences. Inside the **Preferences** dialog box, from the list on the left side, expand the *Java* option, then the *Editor* option. Click on *Content Assist* and ensure there is a check against *Add import instead of qualified name* and against it's sub-option *Use static imports....* Click *OK*. This should enable correct import statements to be automatically generated in your classes.

0.3.2 Initial project set-up

From the menu bar, select File, New, Other, then select *Repast Simphony Project* in the Repast Simphony folder, inside the **Select a wizard** dialog box. Click *Next*. This opens the **Repast Simphony Project** dialog box where you enter the *Project name*. Type in the project name as a string with no spaces: `GridGameOfLife`. Capitalising the initial letter in each word in the project name makes it a little earlier to read, if you have a longer project name. Click *Finish*.

The project is created in the **Package Explorer** window. However, two Groovy classes have been created by default and they will not be used in this project, so they should be deleted. Expand the **src** folder in the project, then expand the package **GridGameOfLife**, and delete the two classes.

We are now ready to create the three classes needed for the project: two agent classes (*Dead* and *Living*) and the context builder class (*GridGoL*). To create the *Dead* class, right click on the package (not the top level project) **GridGameOfLife**, then select New, then Class. The **New Java Class** dialog box will open, and the cursor should already be sitting in the *Name* field. To create the *Dead* class, simply type `Dead` in the field. Click *Finish*. Repeat this process for the *Living* class entering `Living` as the name.

Creating the context builder class is slightly more complicated because we need to instruct Repast that this is the context builder at the time we

create the class. Right click the package again, and select New, then Class as we did for the two agent classes. After entering the name `GridGoL`, click the *Add* button next to the Interfaces field. This opens up a dialog box called **Implemented Interfaces Selection**. The cursor should be sitting in the *Choose interfaces* field. Type `contextbuilder` in that field, and the *Matching items* field will show an interface match called *ContextBuilder - repast.simphony.dataLoader*. Click on that matching item so it is highlighted, and then click *OK*. In the previous dialog box, that interface is now shown in the *Interfaces* field. Click *Finish*.

0.3.3 Constructing the context

The context is the environment in which the agents will operate, so I find it useful to create the context first. However, the order in which the classes and interface are constructed should be based on your preference, as there is no "right" way to sequence the programming. However, it is possible to debug the context builder without having the agent classes finished, but debugging the agents requires the context builder to be completed.

This section only deals with the **GridGoL** class.

We will now build our context. In line 5 of the class, the name and `<T>` are underlined in red, indicating a problem. Replace the `T` with `Object` so that line 5 looks like:

```
public class GridGoL implements ContextBuilder<Object>
```

If you use `Enter` on your keyboard at the end of the line, after that open bracket, your cursor should appear correctly indented on the following line.

The context build is the next step. This:

- defines the underlying and visualisation projections (grid, continuous, or GIS) for the project
- defines the process by which agents will be added to the project, both at initialisation and during the simulation when new agents are added
- defines and sets starting values for the parameters of the simulation. Examples of parameters are the number of each type of agent, and the size of the spaces in which the agents will behave.

This simulation uses only one projection, which is a grid, because agents just need to be defined in terms of x,y locations and co-locate on the same projection. The grid is set to 50 x 50 cells to give a simulation that is large enough for various patterns to emerge. The percentage of the grid that will be initially filled by living agents is set by the user through the simulation GUI, but needs to be parameterised in this class.

To start the context build method, which leads into the following subsections, use `Enter` again so you are on line 7, and type the following:

```
public Context build(Context<Object> context)
context.setId("GridGameOfLife");
```

The `set.Id` must be the same as the project name. If a different text string is used, for example through a typo, the simulation will error at initialisation. The same error outocme is also likely to occur if the `set.Id` is not specified.

If you typed those two lines in, you'll have two close braces "}" at the end of the class, on lines 10 and 12. These close the two methods. If one close brace is missing (e.g. because you pasted the code), add it in.

The red underlined error on the build method starting line has occurred because the method currently does not have the context returned. Fix this by inserting this code between the two close braces:

```
return context;
```

The error showing on `return` will be fixed once some code is added in.

The rest of the code is placed after the `set.Id` line and before the `first` end brace.

Create the underlay and grid projections

The next step is to grid projection. This code defines the factory, which is named `gridFactory`:

```
GridFactory gridFactory = ↙
    ↳ GridFactoryFinder.createGridFactory(null);
```

The projections can be given any suitable name, in this tutorial it is simply called *grid* to have the name clearly associated with the type of projection:

- the string name used (*"grid"*) is the same as the projection name. Although this equivalence is not required, having the same name used reduces the probability of using an incorrect string in other parts of the project.
- the border rule is *WrapAroundBorders()*, so that the edges are contiguous with each other. An agent in a cell on the edge is affected by its neighbours on the edge of the opposite side. This operationalisation is slightly different to the original board-and-token based Game of Life, but means that the agents in cells on the edge act equivalently to other agents that are not situated on an edge.
- *SimpleGridAdder* is the adder used to add agents to the projections. An alternative is to use *RandomGridAdder* for the Living agents, but this causes problems during the simulations when a Dead agent needs to be replaced by a Living agent. Even though the projection cell for the replacement agent is identified with exact x,y co-ordinates, if *RandomGridAdder* is used, this algorithm appears to override the *.moveTo* command. Occasionally the agent replacement will fail, with

a warning written to the console window. Use of *SimpleGridAdder* makes the placement of the initial randomised set-up of agents slightly more complicated - the context builder now requires a set of code to do this instead of randomised placement occurring automatically as a feature of *RandomGridAdder* - but it prevents the agent replacement issue.

- the boolean parameter "multi" is set to *false* so that only one agent can occupy each projection cell.
- the two values at the end are the dimensions of the projection in x,y space as the projection is two dimensional. These have been set to 50 in the simulation, as that provides a model of sufficiently large size that the range of Game of Life patterns can be seen.

The code for the adder is:

```
Grid<Object> grid = gridFactory.createGrid("grid", context,
new GridBuilderParameters<Object>(new WrapAroundBorders(),
new SimpleGridAdder<Object>(), false, 50, 50));
```

Randomly add in the Live agents

Game of Life starts off with a random distribution of agents across the grid. I've chosen to add the Live agents first, although this choice is arbitrary.

The first step is to define the number of agents that are to be added. The value must be an integer. This code creates a counter, that will be used to construct 25% of available agents as Living. It is simply the grid size divided by 4:

```
int livingCount = (int) (50 * 50) / 4;
```

The final step is to add the Living agents to the grid projection:

```
for (int i = 0; i < livingCount; i++) {
 Living liveCell = new Living(grid);
 context.add(liveCell);
 int x = RandomHelper.nextIntFromTo(0, 50 - 1);
 int y = RandomHelper.nextIntFromTo(0, 50 - 1);
 while (!grid.moveTo(liveCell, x, y)) {
  x = RandomHelper.nextIntFromTo(0, 50 - 1);
  y = RandomHelper.nextIntFromTo(0, 50 - 1);
  }
 }
```

Eclipse doesn't automatically import *RandomHelper* so you'll see this underlined in red, and a dialog box of options will appear if you mouse-over that class name. Import 'RandomHelper' from repast.simphony.random and this should be the first option in the list. Click on the hyperlink and the class will be correctly imported.

The first line simply repeats the process for each Living agent. Because the counter starts at zero, the loop must stop when it reaches a count one less than *livingCount*. The last part of that line simply increments i by one each time after loop is started. The second line creates a new Living agent in the grid projection. The third line inserts the Living agent into the context. The last five lines generate random x,y coordinates of integer type and then move the agent to those coordinates, so long as the x,y coordinate is empty. If the x,y coordinate is not empty, a new set of x,y coordinates is generated.

Randomly add in the Dead agents

The Dead agents are now added, filling the remaining empty x,y coordinates. First, the correct number of Dead agents is calculated:

```
int deadCount = (int) (50 * 50) - livingCount;
```

As for the Living agents, the Dead agents are now added to the context and projection. This code moves through all x,y coordinates on the projections, only adding an agent if the coordinate is empty. This fills by columns (x) then rows (y), checking each one in turn. If the selected coordinate is not empty, then the x coordinate is retained, the y coordinate is incremented by 1 and that coordinate is attempted. The process is repeated until all Dead agents are added.

```
Dead deadCell = new Dead(grid);
context.add(deadCell);
for (int x = 0; x < 50 && deadCount > 0; x++) {
 for (int y = 0; y < 50 && deadCount > 0; y++) {
  if (grid.moveTo(deadCell, x, y)) {
   deadCount--;
   if (deadCount > 0) {
         deadCell = new Dead(grid);
         context.add(deadCell);
         }
   }
  }
}
```

The context builder code is now complete, and should look like this:

```
package GridGameOfLife;

import repast.simphony.context.Context;
import repast.simphony.context.space.grid.GridFactory;
import repast.simphony.context.space.grid.GridFactoryFinder;
import repast.simphony.dataLoader.ContextBuilder;
import repast.simphony.random.RandomHelper;
import repast.simphony.space.grid.Grid;
import repast.simphony.space.grid.GridBuilderParameters;
import repast.simphony.space.grid.SimpleGridAdder;
import repast.simphony.space.grid.WrapAroundBorders;
```

```
public class GridGoL implements ContextBuilder<Object> {

public Context build(Context<Object> context) {
 context.setId("GridGameOfLife");

 GridFactory gridFactory = ↙
     ↳ GridFactoryFinder.createGridFactory(null);

 Grid<Object> grid = gridFactory.createGrid("grid", context,
  new GridBuilderParameters<Object>(new WrapAroundBorders(),
   new SimpleGridAdder<Object>(), false, 50, 50));

 int livingCount = (int) (50 * 50) / 4;

 for (int i = 0; i < livingCount; i++) {
  Living liveCell = new Living(grid);
  context.add(liveCell);
  int x = RandomHelper.nextIntFromTo(0, 50 - 1);
  int y = RandomHelper.nextIntFromTo(0, 50 - 1);
  while (!grid.moveTo(liveCell, x, y)) {
   x = RandomHelper.nextIntFromTo(0, 50 - 1);
   y = RandomHelper.nextIntFromTo(0, 50 - 1);
   }
  }

 int deadCount = (int) (50 * 50) - livingCount;
 Dead deadCell = new Dead(grid);
 context.add(deadCell);
 for (int x = 0; x < 50 && deadCount > 0; x++) {
  for (int y = 0; y < 50 && deadCount > 0; y++) {
   if (grid.moveTo(deadCell, x, y)) {
    deadCount--;
    if (deadCount > 0) {
     deadCell = new Dead(grid);
     context.add(deadCell);
     }
    }
   }
  }

 return context;
 }
}
```

Optional: test the context builder

It is possible at this point to test the syntax of the context builder class to debug any issues with this one class. This involves inserting the minimum possible code in the Living and Dead agent classes, so that the simulation will initialise. (It can also run, but the initial state will remain unchanged for all time ticks.) The *Dead* class currently contains only the following code:

```
package GridGameOfLife;


public class Dead {

}
```

and the *Living* class is the same, barring the class name:

```
package GridGameOfLife;


public class Living {

}
```

Eclipse will not run Java code unless there are no errors in the code. There are three lines in the context builder class which have errors. These are the code lines with errors (not sequential lines in the code):

```
Living liveCell = new Living(grid);
Dead deadCell = new Dead(grid);
deadCell = new Dead(grid);
```

and the problem in each case is an undefined constructor due to the lack of code in the *Dead* and *Living* classes. The quick fix, to enable the simulation to initialise, is to simply get Eclipse to automatically create the constructors. This is the second option in the dialog box that appears when you do a mouse-over the error. The outcome of creating the two constructors is that the *Dead* class becomes:

```
package GridGameOfLife;

import repast.simphony.space.grid.Grid;

public class Dead {

 public Dead(Grid<Object> grid) {
  // TODO Auto-generated constructor stub
 }
}
```

and the *Living* class becomes:

```
package GridGameOfLife;

import repast.simphony.space.grid.Grid;

public class Living {

 public Living(Grid<Object> grid) {
  // TODO Auto-generated constructor stub
 }
}
```

The last step is to edit the *context.xml* file so that the context builder class we just created is used in the simulation. This file is accessible in Package Explorer: expand **GridGameofLife.rs**, which is the folder immediately after the libraries. The file we are editing is the first in the list. Double click to open it, and add the following line before the close context command:

```
<projection type="grid" id="grid"></projection>
```

The projection type must match that used in the context builder, and the id must match the string name in the context builder. The context.html file should now look like:

```
<context id="GridGameOfLife"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation= ⬋
    ↳ "http://repast.org/scenario/context">
<projection type="grid" id="grid"></projection>
</context>
```

Save all files if you haven't already done so.

Run the model.

The Repast Simphony GUI now appears. This requires some editing before the model can be initialised.

First, the correct context builder must be referenced. In **Scenario Tree**, expand *Data Loaders* and delete the default data loader called *XML & Model Init Context Builder* as this will be replaced by proper one. Right click *Data Loaders*, select the only option *Set Data Loader* then choose the *Custom ContextBuilder Implementation* option, which is the third one down the list. Click *Next*. The dialog box will now show the context builder class name for the project. Click *Next* then *Finish*. The correct context builder class name is now shown under *Data Loaders* in the **Scenario Tree**. I *Save* at this point.

Second, the visualisation must be defined. Still in **Scenario Tree**, right click *Displays* and select the only option *Add Display*. Choose a display name, I chose "GoLDisplay". Arrow the only option, *grid* into the right hand box. Click *Next*. Move the *Dead* and *Living* agent types over to the right side. Click *Next*. The next step is just to select the colour and shape formatting for the *Dead* and *Living* agents. I retained the circle shape, but selected a green colour for the *Living* agents and a black colour for the *Dead* agents. Once you are happy with your colour and shape choices, click *Next*. I choose a white **Grid color** to distinguish it from the *Dead* agents for debugged purposes. Click *Next* and then *Finish*. In the **Scenario Tree**, there should now be a display (with your display name) shown under *Displays*. I do a second *Save* at this point.

The simulation initialisation is now ready to test. When you click the initialise button, a 50x50 grid show display, filled with a mixture of *Dead* and *Living* agents. If you're unhappy with your display colour choices and

want to redo them, or anything else in the display, then you can simply double-click on the display name. A dialog box will open, which then allows you to make changes to the display options without needing to redo the display from scratch. You will need to reset the run to do this, otherwise the **Scenario Tree** items will be greyed out. I also *Save* at this point.

0.3.4 Constructing the Dead agent class

This program uses the rules stated on the Wikipedia page for the Game of Life, so a *Dead* agent will be replaced by a *Living* agent if that cell has exactly three living neighbours in its Moore neighbourhood. Because both agent classes need to determine the state of their neighbours, then both potentially change state for the next time, scheduling annotations are used to ensure that processes occur in the correct order. There are two scheduling annotations in each class, one to check the Moore neighbour, and one to remove the current agent and replace with one of the other class (if relevant). If the agent class does not need to change, the agent is unaffected by that time step in the simulation.

Immediately after:

```
public class Dead {
```

Insert the following lines:

```
private Grid<Object> grid;
private int state;
```

These lines simply allow these variables to be used anywhere inside the *Dead* class.

When we tested the context builder code, we made Eclipse add in some useful code to the class:

```
public Dead(Grid<Object> grid, Grid<Object> underlay) {
 // TODO Auto-generated constructor stub
 }
```

We are going to create the constructor, so delete the commented line starting // TODO. If you didn't test the context builder, you will need to add in the relevant constructor lines. Amend the constructor so it looks like this:

```
public Dead(Grid<Object> grid) {
 this.grid = grid;
 }
```

The purpose of this added line is so that the *grid* object can be passed as an object to methods later in the class.

The next part of the code will examine the Moore neighbourhood surrounding each *Dead* agent to determine if any of them should be replaced, on a one-for-one basis, with a *Living* agent. A ScheduledMethod annotation is used to correctly sequence the examination and replace methods. The

higher the priority assigned, the earlier the method related to the annotation will run in the time step. This simulation makes each *Dead* agent method run immediately before its corresponding *Living* agent method. There are 4 annotations, so the highest priority=4. The annotations start in the first time step, and run once each time step. The code below ensures that the check of the Moore neighbourhood for the *Dead* agents always occurs as the first method each time step.

```
@ScheduledMethod(start = 1, interval = 1, priority=4)
```

This part of the code gets the x,y location of the *Dead* agent. It then queries the Moore neighbourhood around that agent. Each time it finds a *Living* agent in that neighbourhood, a counter is incremented by 1. If there are exactly 3 *Living* neighbours, state is set to 1, otherwise state is set to 0. A state of 1 is used to identify agents that must be *Living* at the next time step, and 0 identifies agents that must be *Dead*. The states are used in the next method, which occurs at a later time.

```
public void step1() {
 MooreQuery<Dead> query = new MooreQuery(grid,this);
 int neighbours = 0;
 for (Object o : query.query()){
  if (o instanceof Living) {
   neighbours++;
   }
 }
 if (neighbours==3) {
   state=1;
   }
 else {
  state=0;
  }
}
```

The last step is to replace the relevant *Dead* agents with *Living* ones, on the basis of the state value. The annotation is used to make the *Dead* replacement occur as the last method in each time step, hence why priority is equal to 1.

```
@ScheduledMethod(start = 1, interval = 1, priority=1)
```

The following code only replaces the relevant *Dead* agents. *Dead* agents who are to remain *Dead* for the next time step are ignored. For each *Dead* agent to be removed:

- the x,y coordinates of the agent in the grid projection is retrieved.
- the agent is removed from the context.
- a *Living* agent is constructed.
- the new *Living* agent is added to the context and to the x,y coordinates of the removed *Dead* agent in the grid projection.

```
public void step2() {
 if (state==1) {
  GridPoint gpt = grid.getLocation(this);
  Context<Object> context = ContextUtils.getContext(this);
  context.remove(this);
  Living livingCell = new Living(grid);
  context.add(livingCell);
  grid.moveTo(livingCell, gpt.getX(), gpt.getY());
  context.add(livingCell);
  }
 }
```

The *Dead* class code is now complete and should look like this:

```
package GridGameOfLife;

import java.util.Iterator;

import repast.simphony.context.Context;
import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.query.space.grid.MooreQuery;
import repast.simphony.space.grid.Grid;
import repast.simphony.space.grid.GridPoint;
import repast.simphony.util.ContextUtils;

public class Dead {
 private Grid<Object> grid;
 private int state;

 public Dead(Grid<Object> grid) {
  this.grid = grid;
 }

 // calculate the state for the next time tick for dead cells
 @ScheduledMethod(start = 1, interval = 1, priority = 4)
 public void step1() {
  MooreQuery<Dead> query = new MooreQuery(grid, this);
  int neighbours = 0;
  for (Object o : query.query()) {
   if (o instanceof Living) {
    neighbours++;
   }
  }
  if (neighbours == 3) {
   state = 1;
  } else {
   state = 0;
  }
 }

 // visualise the change into the grid
 @ScheduledMethod(start = 1, interval = 1, priority = 1)
 public void step2() {
  if (state == 1) {
   GridPoint gpt = grid.getLocation(this);
```

```
    Context<Object> context = ContextUtils.getContext(this);
    context.remove(this);
    Living livingCell = new Living(grid);
    context.add(livingCell);
    grid.moveTo(livingCell, gpt.getX(), gpt.getY());
    context.add(livingCell);
   }
  }
}
```

0.3.5 Constructing the Living agent class

The code is just a mirror of the Dead agent class with the following changes:

- the @ScheduledMethod priority for querying the Moore neighbourhood is 3.
- the rule for replacing a *Living* agent will a *Dead* agent is that the cell does not have either 2 or 3 living neighbours in its Moore neighbourhood.
- the @ScheduledMethod priority for replacing a *Living* agent with a *Dead* one is 2.

The *Living* class code is now complete and should look like this:

```
package GridGameOfLife;

import java.util.Iterator;

import repast.simphony.context.Context;
import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.query.space.grid.MooreQuery;
import repast.simphony.space.grid.Grid;
import repast.simphony.space.grid.GridPoint;
import repast.simphony.util.ContextUtils;

public class Living {
 private Grid<Object> grid;
 private int state;

 public Living(Grid<Object> grid) {
  this.grid = grid;
 }

 // calculate the state for the next time tick for living cells
 @ScheduledMethod(start = 1, interval = 1, priority = 3)
 public void step1() {
  MooreQuery<Living> query = new MooreQuery(grid, this);
  int neighbours = 0;
  for (Object o : query.query()) {
   if (o instanceof Living) {
```

```
      neighbours++;
     }
    }
    if (neighbours == 2 || neighbours == 3) {
     state = 1;
     } else {
      state = 0;
     }
    }

   // visualise the change into the grid
   @ScheduledMethod(start = 1, interval = 1, priority = 2)
   public void step2() {
    if (state == 0) {
     GridPoint gpt = grid.getLocation(this);
     Context<Object> context = ContextUtils.getContext(this);
     context.remove(this);
     Dead deadCell = new Dead(grid);
     context.add(deadCell);
     grid.moveTo(deadCell, gpt.getX(), gpt.getY());
     context.add(deadCell);
    }
   }
}
```

0.3.6 Running the full model

The model is now finished and ready to run. If you wish to set a maximum number of time steps, or slow down the transition between time steps, use the **Run Options** tab at the bottom of the **Scenario Tree**. The run options are not saved between run sessions.

0.3.7 Showing the count of agents at each time step

For all the objects being created here: data set; graph, text file, these can be amended after creation by simply double-clicking on the relevant object in the **Scenario Tree**.

A record of the count of *Living* agents and *Dead* agents can be collected from the simulation. In the **Scenario Tree**, right-click on *Data Sets*, and select the only option *Add Data Set*. This action brings up the **Data Set Editor** dialog box. Fill in a name for the *Data Set Id*, I used "AgentCount", and keep the "Aggregate" option, as percentage is an aggregate measure. Click *Next*.

The data to be collected is the count of the *Living* and *Dead* agents that exist at each time point in the simulation, so make sure that the "Tick Count" option is selected in the *Standard Sources* tab. This should be the default tab view at this point, if not then simply select that tab.

The *Method Data Sources* is the second tab, and click on this. The information in this tab is used to construct the aggregate count. Click on the *Add* button at the bottom of the tab. Do this twice, this will add in two data source rows. You need to add in two data sources, one for the *Living* agents and one for the *Dead* agents. One of the added rows may be incorrect (the context builder class was added in as the first row in my set-up). To edit any fields in the *Method Data Sources*, you must **double-click** in the cell you wish to modify.

The two rows in this tab should be set up with the following settings, note that the sequence does not matter. Do **not** attempt to change the Method cell, as this will cause errors in your console.

Source Name	Agent Type	Method	Aggregate Operation
Dead Count	Dead	N/A	Count
Living Count	Living	N/A	Count

Table 1: Method Data Sources table.

Click *Next*. The default **Schedule Parameters** options should be retained. The counts will be collected at each time step, at the end of the time step. Click *Finish*. You should now see *AgentCount* under *Data Sets* in the **Scenario Tree**. I save at this point.

While the counts are now being created, currently there is no place for them to be stored. They can be output to an external *File Sink* for use in other software (for example), and they can be recorded within Repast Simphony. We're going to create a graph in Repast Simphony to view the counts over time, and also create a text file to store the data for possible use elsewhere. This is all done within the **Scenario Tree**.

To create a graph, right-click on *Charts* and select *Add Time Series Chart*. The **Time Series Editor** dialog box appears. Give your graph a name, I used "Counts over Time". The name of your data set is auto-populated into the *Data Set* field. Click *Next*. The **Configure Chart Data Properties** screen lets you select the data series, and associated legend and colour, to display in the graph. Again, you can configure some of the cells by double-clicking them. I changed the legends to simply *Dead* and *Living*, and changed their colours to black and green respectively (because those were the agent colours I choose for the visualisation in the projections). Click *Next*. The final step is to adjust the **Configure Chart Properties** to the properties you wish to have. I gave it a title (Agent Counts), changed the *X-Axis* label to "Time" and entered a *Y-Axis* label of "Number of Agents". The other properties can also be amended as you wish, I retained their defaults. Click *Finish*. I save at this point.

To create an external text file, in the **Scenario Tree**, right-click on *Text Sinks*, and select the option *Add File Sink*. The **File Sink Editor** dialog box appears. Enter a useful name that will be displayed in the **Scenario**

Tree, I chose "Agent Count Data". Our one data set is auto-populated into the *Data Set ID* field. Still on this screen, the last step is to select the data we wish to record, and the column order that it will appear in the file. Click on the data item (e.g. *Tick*), which then activates the green arrow, and put all 3 into the right-hand box. Having the *Tick Count* as the first column is logical, the column sequence of the counts is a matter of personal preference. To re-order items in that right-hand box, click on the item you wish to move and then use the up and down arrows to reposition it. Click *Next*. The name and format of the output file can now be defined. We can retain the defaults, which will produce a file in csv format. A handy feature is that Repast Simphony autosaves the names of variables as a header row in the text file output. The auto-selected option of *Insert Current Time into File Name* is very useful as it means that output files will not be overwritten. This is useful in situations like debugging or comparing outputs using different simulation settings - it means that you don't have to remember to change the output file name each time you run. Click *Finish*. I save at this point.

Because data is now being collected, and there is no defined endpoint to the simulation, it is useful to enter a stop time for the simulation. This method does not retain the changes from one run to the next, because it only sets the options for the current run. In the tabs at the bottom right of **Repast Simphony**, click on the **Run Options**. In the *Stop At* field, enter the maximum number of ticks you wish the simulation to use. The longer the simulation time, the wider the graph and the longer the text file output. I entered 500.

You can now initialise your model and run it. When the simulation finishes, you will have two tabs on the right-hand side. The first tab is the visualisation of the simulation. The second tab, which is created because there is a graph associated with the simulation, shows the agent counts over time. Just click on the tab to view the graph (it's not viewable through the **Scenario Tree**, that is just set-up information). There is also a text file that has been created with the data for each time step. By default, Repast Simphony saves the file into the main package folder in the workspace. Note that because the model initialisation time is set to 0, rather than 1, there is an "extra" row of data in the graph and output file.