Space Details

Key:	SIM
Name:	Repast Simphony
Description:	
Creator (Creation Date):	turtlebender (Apr 26, 2006)
Last Modifier (Mod. Date):	etatara (Mar 18, 2009)

Available Pages

Repast Simphony Reference

This page last changed on Nov 30, 2007 by collier.

Adding Agent Adaptation Using Genetic Algorithms

This page last changed on Aug 26, 2008 by north.

Repast Simphony (Repast S) uses the Java Genetic Algorithms Package (JGAP) (http://jgap.sourceforge.net/) to implement genetic algorithms. JGAP can be used directly within agents or the Repast S genetic algorithms (GA) wrapper can be used. Directly using JGAP provides maximum flexibility but requires more complex programming. The Repast S GA wrapper simplifies the use of JGAP while preserving most options. It is recommended that Repast S users start with the Repast S wrapper and, only if needed, eventually move to directly use of JGAP. The Repast S regression wrapper is consistent with direct use of JGAP so this should relatively straightforward. Obviously, as with all machine learning tools, it is important to make sure that genetic algorithms, in general, and the chosen formulation, in particular, are appropriate for your application.

Documentation on how to use JGAP directly can be found on their web site, http://jgap.sourceforge.net/ In particular, please see the following page:

http://jgap.sourceforge.net/doc/tutorial.html

To use the Repast S GA wrapper in the visual editor simply follow these steps:

1.) Add a property of type "Genetic Algorithm" to an agent. "evaluate" is the name of the fitness function that needs to be created as described in step two. "populationSize" is the count of the members in the GA population. The "Gene" array gives the set of genes, either "IntegerGene" or "DoubleGene," to be used. The parameters to the constructors (e.g., "min1" and "max1") are the optional lower and upper bounds for that gene. The template is as follows:

new RepastGA(this, "evaluate", populationSize, new Gene[]{ new IntegerGene(min1, max1) new DoubleGene(min2, max2)})

2.) Add a behavior that takes a double precision array (i.e., "double[]"), returns an positive integer result, and that uses the fitness function name given above (e.g., "evaluate"). The double array is the current population member to be evaluated. The genes are in the same order as the template given in step one. Higher fitness result values mean greater fitness levels. The exact range does not matter as long as the numbers are positive.

3.) Using the Wizard dialog, chose "Advanced Adaptation," "Use a Genetic Algorithm," "Get the Best Solution from a Genetic Algorithm," and then fill in the form to get a solution. The returned double array is the current best solution. The genes are in the same order as the template given in step one.

4.) If the population needs to be reset, perhaps due to a plateau, using the Wizard dialog, chose "Advanced Adaptation," "Use a Genetic Algorithm," "Reset the Population of a Genetic Algorithm," and then fill in the form.

To use the Repast S GA wrapper in the Java or Groovy simply follow these steps:

1.) Add an agent class field of type RepastGA and create an instance following this template:

new RepastGA(this, "evaluate", populationSize, new Gene[]{ new IntegerGene(min1, max1) new

DoubleGene(min2, max2)})

"evaluate" is the name of the fitness function that needs to be created as described in step two. "populationSize" is the count of the members in the GA population. The "Gene" array gives the set of genes, either "IntegerGene" or "DoubleGene," to be used. The parameters to the constructors (e.g., "min1" and "max1") are the optional lower and upper bounds for that gene.

2.) Add a class method behavior that takes a double precision array (i.e., "double[]"), returns an positive integer result, and that uses the fitness function name given above (e.g., "evaluate"). The double array is the current population member to be evaluated. The genes are in the same order as the template given in step one. Higher fitness result values mean greater fitness levels. The exact range does not matter as long as the numbers are positive. For higher performance, advanced users can implement org.jgap.FitnessFunction in place of just the creating the simple "int evaluate(double[])" method.

3.) Use the following method template to get a solution:

double[] solution = model.getBestSolution(cycles)

"cycles" is the number of evolutionary steps to use to compute the requested solution. The returned double array is the current best solution. The genes are in the same order as the template given in step one.

4.) If the population needs to be reset, use the following method template:

model.reset()

Adding Agent Adaptation Using Neural Networks

This page last changed on Aug 25, 2008 by north.

Repast Simphony (Repast S) uses the Java Object Oriented Neural Engine (Joone) (http://www.jooneworld.com/) library to implement neural networks. Joone can be used directly within agents or the Repast S neural network wrapper can be used. Directly using Joone provides maximum flexibility but requires more complex programming. The Repast S wrapper simplifies the use of Joone while preserving most options. It is recommended that Repast S users start with the Repast S wrapper and, only if needed, eventually move to directly use of Joone. The Repast S wrapper is consistent with direct use of Joone so this should relatively straightforward. Obviously, as with all machine learning tools, it is important to make sure that neural networks, in general, and the chosen network design, in particular, are appropriate for your application.

Documentation on how to use Joone directly can be found on their web site, http://www.jooneworld.com/

To use the Repast S neural network wrapper in the visual editor simply follow these steps:

1.) Add a property of type "Linear Neural Network," "Logistic Neural Network," or "Softmax Neural Network" to an agent to produce the corresponding type of neural network. Replace the integer (i.e., int) array elements with a description of the layers for your application. The first number in the array is the count of input neurons, the last number is the count of output neurons, and the intermediate numbers are the interstitial layers. You can have as many interstitial layers as you would like (i.e., zero or more).

2.) Using the Wizard dialog, chose "Advanced Adaptation," "Use a Neural Network," "Train a Neural Network Model," and then fill in the form to store data for fitting. The input data array is indexed by the number of training examples and the number of input neurons. The output data array is indexed by the number of training examples and the number of output neurons.

3.) Using the Wizard dialog, chose "Advanced Adaptation," "Use a Neural Network," "Forecast from a Neural Network," and then fill in the form to get a forecast. The input data array is indexed by the number of input neurons. The output data array is indexed by the number of output neurons.

To use the Repast S regression wrapper in the Java or Groovy simply follow these steps:

1.) Add an agent class field of type NeuralNet and create an linear, logistic, or softmax instance as follows:

JooneTools.create_standard(new int[]{inputNeurons intermediateLayer1 intermediateLayer2 intermediateLayerN outputNeurons}, JooneTools.LINEAR)).

JooneTools.create_standard(new int[]{inputNeurons intermediateLayer1 intermediateLayer2 intermediateLayerN outputNeurons}, JooneTools.LOGISTIC)).

JooneTools.create_standard(new int[]{inputNeurons intermediateLayer1 intermediateLayer2 intermediateLayerN outputNeurons}, JooneTools. SOFTMAX)).

2.) Use the following method to store data for fitting where "model" is the new field name:

```
JooneTools.train(model, inputDataArray, targetOutputDataArray, epochs, convergenceLimit, 0, null);
```

The input data array is indexed by the number of training examples and the number of input neurons. The output data array is indexed by the number of training examples and the number of output neurons. The model will be trained until "epochs" cycles are completed or the "convergenceLimit" of root mean square error is achieved, whichever comes first.

3.) Use the following method to get a forecast where "model" is the new field name:

```
double outputData[] = JooneTools.interrogate(model, inputData)
```

The input data array is indexed by the number of input neurons. The output data array is indexed by the number of output neurons.

Adding Agent Adaptation Using Regression

This page last changed on Aug 25, 2008 by north.

Repast Simphony (Repast S) uses the OpenForecast (http://openforecast.sourceforge.net/) library to implement regression. OpenForecast can be used directly within agents or the Repast S regression wrapper can be used. Directly using OpenForecast provides maximum flexibility but requires more complex programming. The Repast S regression wrapper simplifies the use of OpenForecast while preserving most options. It is recommended that Repast S users start with the Repast S wrapper and, only if needed, eventually move to directly use of OpenForecast. The Repast S regression wrapper is consistent with direct use of OpenForecast so this should relatively straightforward. Obviously, as with all statistical tools, it is important to make sure that regression, in general, and the chosen regression fit, in particular, are appropriate for your application. The method "getForecastType()" can be used to find a text description of the selected equation for both types of regression. It is recommended that this be checked for appropriateness.

Documentation on how to use OpenForecast directly can be found on their web site, http://openforecast.sourceforge.net/

To use the Repast S regression wrapper in the visual editor simply follow these steps:

1.) Add a property of type "Multilinear Regression Model" or "Best Fit Regression Model" to an agent "Multilinear Regression Model" always fits input data sets using a linear equation for one input variable or multilinear equation for multiple input variables. "Best Fit Regression Model" causes OpenForecast to consider a range of different equation families and then find the best fitting member of the best fitting equation family. This includes linear and multilinear equations as well as other families as defined in the OpenForecast documentation.

2.) Using the Wizard dialog, chose "Advanced Adaptation," "Use a Regression Model," "Store Data for a Regression Model," and then fill in the form to store data for fitting.

3.) Using the Wizard dialog, chose "Advanced Adaptation," "Use a Regression Model," "Forecast from a Regression Model," and then fill in the form to get a forecast. Please note that it generally is necessary to return default forecasts for several steps before enough example data is available for a good quality forecast.

To use the Repast S regression wrapper in the Java or Groovy simply follow these steps:

1.) Add an agent class field of type RepastRegressionModel and create an instance (i.e., new RepastRegressionModel(false)). Set the constructor parameter to "false" for "Multilinear Regression Model" or "true" for "Best Fit Regression Model." "Multilinear Regression Model" always fits input data sets using a linear equation for one input variable or multilinear equation for multiple input variables. "Best Fit Regression Model" causes OpenForecast to consider a range of different equation families and then find the best fitting member of the best fitting equation family. This includes linear and multilinear equations as well as other families as defined in the OpenForecast documentation. Obviously, as with all statistical tools, it is important to make sure that the chosen functions, both multilinear and best fit, are appropriate for your application. The method "getForecastType()" can be used to find a text description of the selected equation for both types of regression. It is recommended that this be checked for appropriateness.

2.) Use the new field's "add(...) method to store data for fitting (i.e., model.add(y, x1, x2, x3)).

3.) Use the new field's "forecast(...)" method to get a forecast (i.e., double x = model.forecast(y1, y2, y3)). Please note that it generally is necessary to return default forecasts for several steps before enough example data is available for a good quality forecast.

Adding Controller Actions

This page last changed on Aug 12, 2009 by etatara.

Controller action can be added and integrated into the controller tree via plugin extension points. The repast.simphony.core plugin defines the following:

```
<extension-point id="composite.action">
    <parameter-def id="creatorClass"/>
</extension-point>
 <extension-point id="component.action">
    <parameter-def id="actionIO"/>
  </extension-point>
```

- "composite.action" is used to create a parent controller action, that is, a controller action that is composed of child controller actions. The value of the "creatorClass" parameter must be the name of a class that implements the CompositeControllerActionCreator interface.
- "component.action" is used to create child or component controller actions that are leaves in the controller action tree. The value of the "actionIO" parameter must the name of a class that implements the ControllerActionIO interface. This class is primarily responsible for serializing and deserializing (saving and loading) a specific type of controller action. For example, the CNDataLoaderControllerActionIO class is responsible for serializing and deserializing the class name data loader action.

and the repast.simphony.gui plugin extends these two extension points with:

```
<extension-point id="composite.action" parent-plugin-id="repast.simphony.core"
parent-point-id="composite.action">
        <parameter-def id="label"/>
  </extension-point>
<extension-point id="component.action" parent-plugin-id="repast.simphony.core"
parent-point-id="component.action">
        <parameter-def id="editorCreator"/>
        <parameter-def id="parentMenuItem" multiplicity="none-or-one"/>
        <parameter-def id="parentID"/>
    </extension-point>
```

These extend the above by adding gui related parameters.

- composite.action adds a label parameter the value of which will be the label for the parent action as it is displayed in the controller action tree.
- component.action adds 3 new parameters
  1. editorCreator. The value of this parameter is a class that implements the ActionEditorCreator interface. The editor created by the class that implements this interface will be used to create a gui editor for editing this action.
  2. parentMenuItem. The value of this parameter is a class that implements the EditorMenuItem. This will create the item in the parent menu that allows actions of this type to be created.
  3. parentID. this is the id of the parent of this controller action.Default parent ids are defined in repast.plugin.ControllerActionConstants:

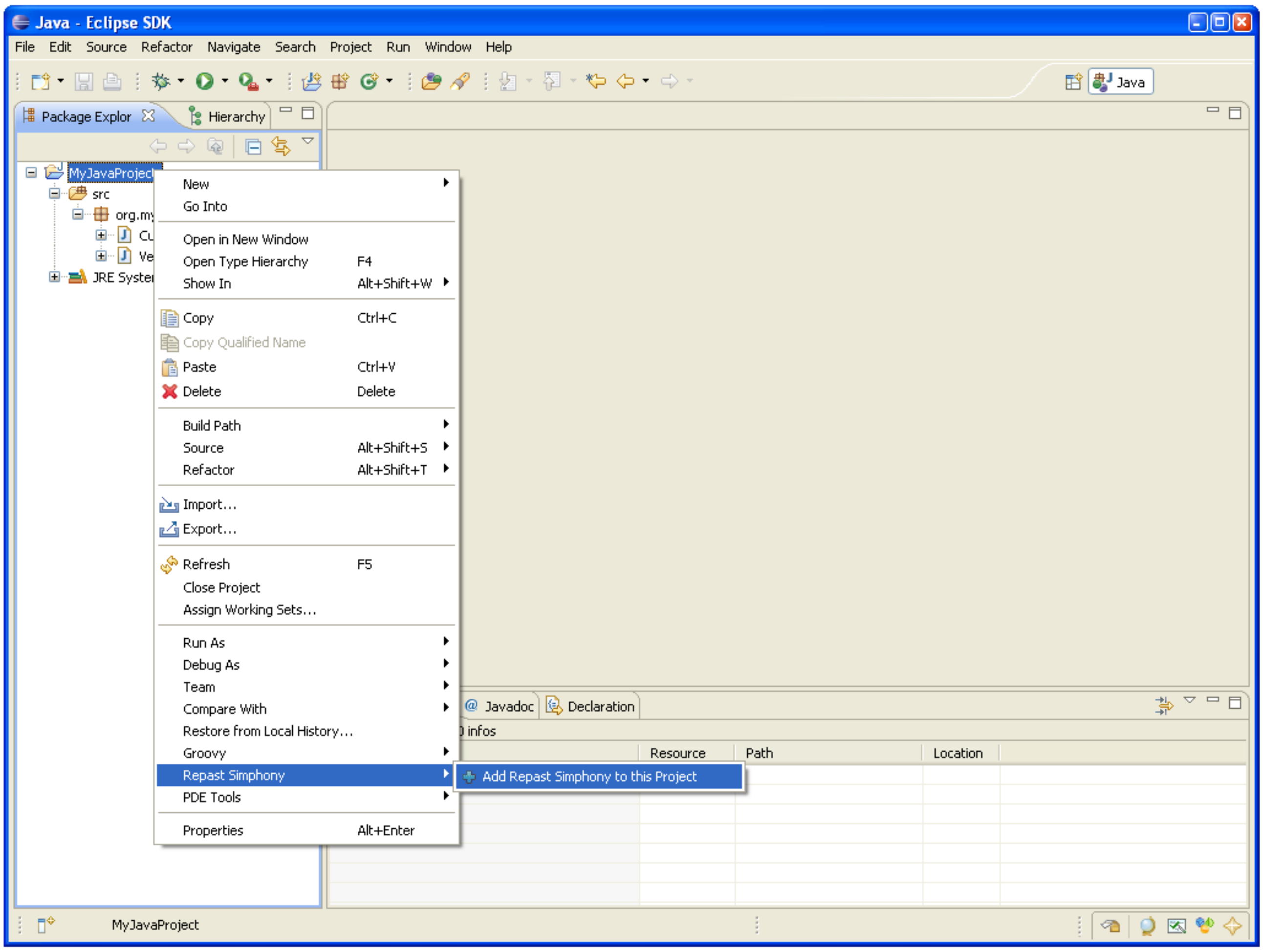
```
String VIZ_ROOT = "repast.controller.action.viz";
String DATA_LOADER_ROOT = "repast.controller.action.data_loaders";
String SCHEDULE_ROOT = "repast.controller.action.schedule";
String OUTPUTTER_ROOT = "repast.controller.action.outputters";
String GRAPH_ROOT = "repast.controller.action.graphs";
String REPORT_ROOT = "repast.controller.action.reports";
String DATA_SET_ROOT = "repast.controller.action.data_sets";
```

All the controller actions in repast simphony are added via this mechanism so examining their plugin.xml files and related code is a good place to learn more.

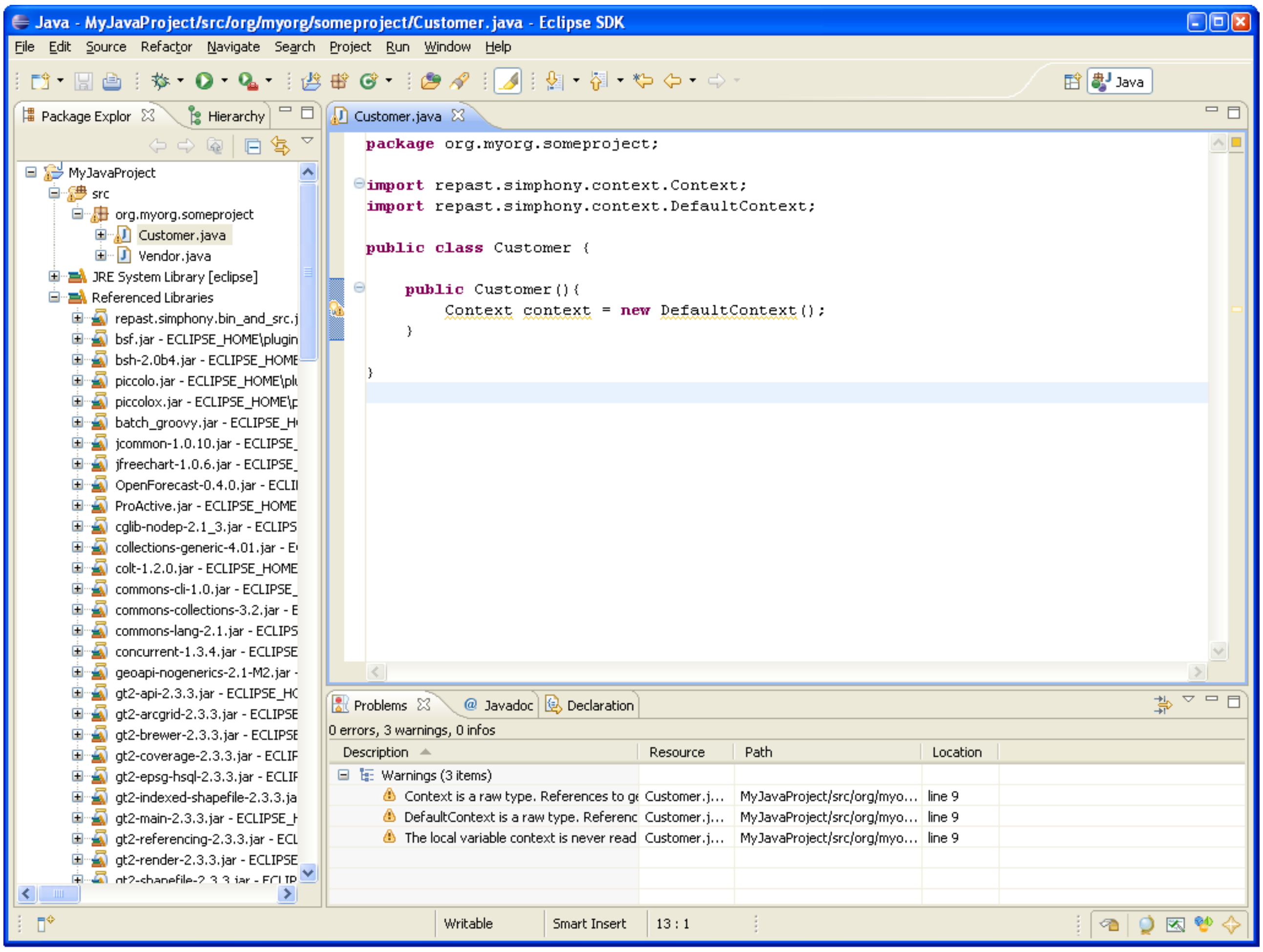
Adding Repast to an Existing Java Project

This page last changed on Nov 22, 2010 by north.

If you have an existing Java project in which you would like to reference Repast classes, you can right click on the project and select Repast Simphony -> Add Repast Simphony to this Project.



This will add the Repast libraries to the project class path so that you can reference Repast classes and run simulations from your project without needing to manually add all of the Repast libraries and dependencies.

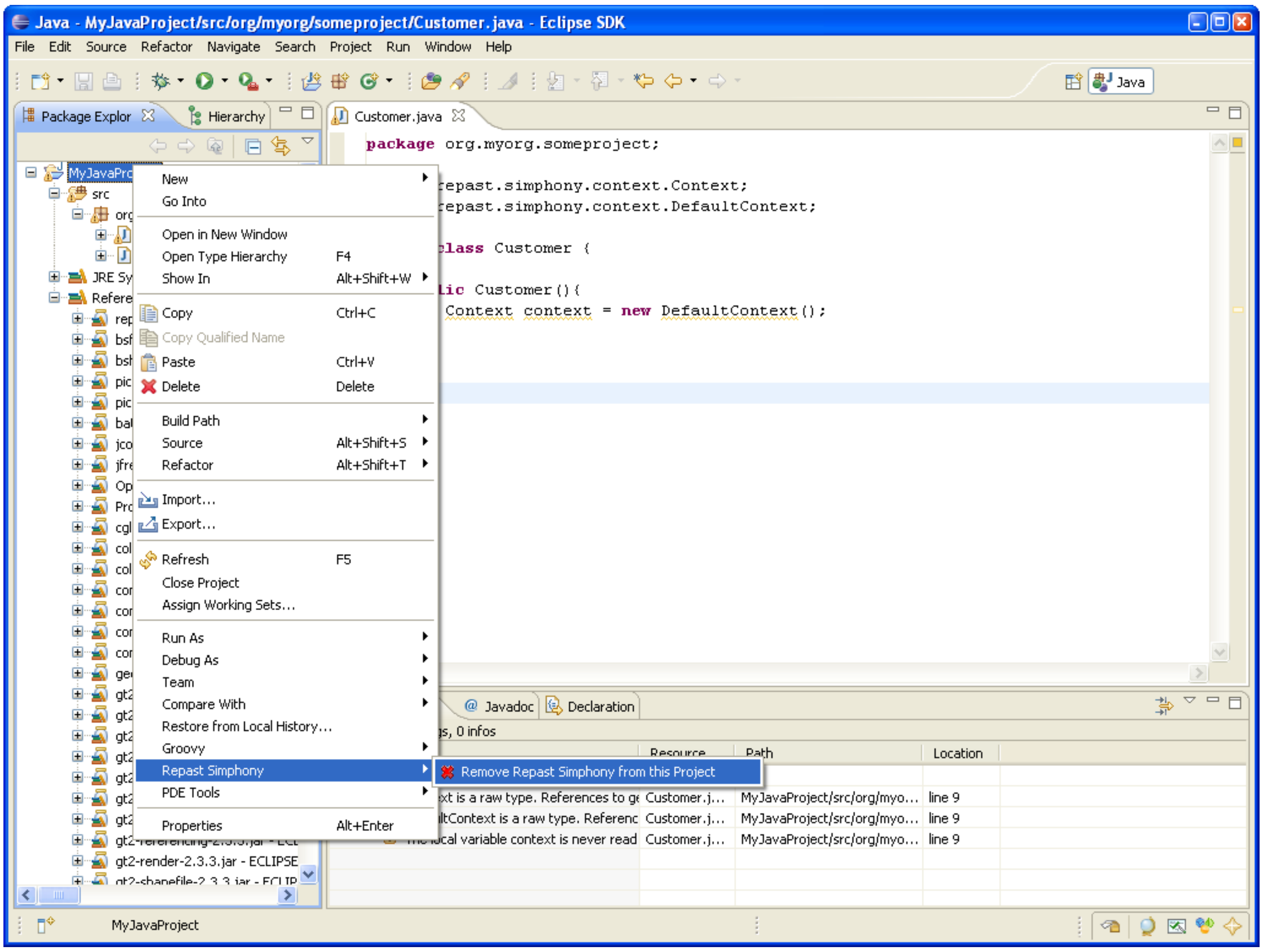

Java - MyJavaProject/src/org/myorg/someproject/Customer.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explor
Hierarchy
MyJavaProject
src
org.myorg.someproject
Customer.java
Vendor.java
JRE System Library [eclipse]
Referenced Libraries
repast.simphony.bin_and_src.j
bsf.jar - ECLIPSE_HOME\plugin
bsh-2.0b4.jar - ECLIPSE_HOME
piccolo.jar - ECLIPSE_HOME\pl
piccolox.jar - ECLIPSE_HOME\p
batch_groovy.jar - ECLIPSE_H
jcommon-1.0.10.jar - ECLIPSE_
jfreechart-1.0.6.jar - ECLIPSE_
OpenForecast-0.4.0.jar - ECLI
ProActive.jar - ECLIPSE_HOME
cglib-nodep-2.1_3.jar - ECLIPS
collections-generic-4.01.jar - E
colt-1.2.0.jar - ECLIPSE_HOME
commons-cli-1.0.jar - ECLIPSE_
commons-collections-3.2.jar - E
commons-lang-2.1.jar - ECLIPS
concurrent-1.3.4.jar - ECLIPSE
geoapi-nogenerics-2.1-M2.jar -
gt2-api-2.3.3.jar - ECLIPSE_HC
gt2-arcgrid-2.3.3.jar - ECLIPSE
gt2-brewer-2.3.3.jar - ECLIPSE
gt2-coverage-2.3.3.jar - ECLIF
gt2-epsg-hsql-2.3.3.jar - ECLIF
gt2-indexed-shapefile-2.3.3.ja
gt2-main-2.3.3.jar - ECLIPSE_I
gt2-referencing-2.3.3.jar - ECL
gt2-render-2.3.3.jar - ECLIPSE
Customer.java
Problems
Javadoc
Declaration
Writable
Smart Insert
13 : 1

```java
package org.myorg.someproject;

import repast.simphony.context.Context;
import repast.simphony.context.DefaultContext;

public class Customer {

    public Customer(){
        Context context = new DefaultContext();
    }

}
```

0 errors, 3 warnings, 0 infos

Description	Resource	Path	Location
Warnings (3 items)			
Context is a raw type. References to ge	Customer.j…	MyJavaProject/src/org/myo…	line 9
DefaultContext is a raw type. Referenc	Customer.j…	MyJavaProject/src/org/myo…	line 9
The local variable context is never read	Customer.j…	MyJavaProject/src/org/myo…	line 9

To remove Repast functionality from the project, select Repast Simphony -> Remove Repast Simphony from this Project.



```
package org.myorg.someproject;

epast.simphony.context.Context;
epast.simphony.context.DefaultContext;

class Customer {

lic Customer() {
 Context context = new DefaultContext();
```

s, 0 infos

	Resource	Path	Location
xt is a raw type. References to ge	Customer.j...	MyJavaProject/src/org/myo...	line 9
ltContext is a raw type. Referenc	Customer.j...	MyJavaProject/src/org/myo...	line 9
ocal variable context is never read	Customer.j...	MyJavaProject/src/org/myo...	line 9

Batch Parameters

This page last changed on Aug 10, 2010 by etatara.

Repast Simphony supports the creation of a parameter sweep through an xml file or through a scripted bsf file. Both define a sweep through the defintion of parameter setters. These setters may be constant setters such that the parameter value remains at its initial value, or the setters may define a list of values to sweep through, or lastly the setter may function as a numeric stepper that incrementally steps from an initial value through some ending value. These setters are organized in tree such that the sweep defined by a child is fully iterated through before the parent is incremented.

1. XML Parameter Format

The root of the xml parameter sweep file is the **sweep** element.

```
<sweep runs="x">
...
</sweep>
```

The runs attribute defines the number of times the sweeper will sweep through the defined space in its entirety.

Individual parameters are identified as **parameter** elements and have two required attributes: **name** and **type**.

```
<parameter name="parameter_1" type=["number", "list", "constant"] ... />
```

- **name** is the name of the parameter. The current value of the parameter can be retrieved from the Parameters object using this name.
- **type** identifies the how the value of the parameter will be set over the course of subsequent batch runs. Depending on the type, the parameter element will have additional attributes.

Parameter Types

Currently 3 types of parameters are supported.

1. **number** -- the number type defines a parameter space with a starting, ending and step value. The sweep mechanism will sweep through the space defined by these values by incrementing the starting value with the step value until the ending value is reached. For example,

   ```
   <parameter name="num_1" type="number" start="1" end="4" step="1"/>
   ```

   will start with a value of 1 and increment by 1 until 4 is reached. The type of number (long, double, float and integer) is inferred from the start, end and step values. If any of the values ends with 'f', for example,

   ```
   4f or 3.21f
   ```

the parameter will be of the float type. Similarly, if any of the values end with 'L', the type will be a long. In the absence of a "f" suffix, if any of the numeric values contain a decimal point and one or more digits (e.g. 4.0) then the type will be a double. Otherwise, the type will be int.

2. **list** -- the list type defines a parameter space as a list.

```
<parameter name="list_val" type="list" value_type=["string", "boolean", "double", "int", "long", "float"] values="x y z"/>
```

   The sweeper will iterate through the values in the list setting the parameter to that value.The **value_type** attribute defines the type of the list elements and can be one of string, boolean, double, int, long, float. Elements are delimited by a blank space. String elements can be surrounded by single quotes (') if they contain blank space. For example,

```
<parameter name="list_val" type="list" value_type="string" values="'foo bar' 'bar' 'baz'"/>
```

   defines 3 elements foo bar, bar and baz.

3. **constant** -- constant types define a parameter space consisting of a single constant value.

```
<parameter name="const_1" type="constant" constant_type=["number", "boolean", "string"]
value="x"/>
```

   The sweeper will set the parameter to this constant value. The **constant_type** attribute defines the type of the constant value. A constant type of "number" works as its does above where the specified type of number (float and so forth) is derived from the value itself.

Nesting Parameters

Parameters can be nested such that the space defined by the child or nested parameter will be swept before incrementing any values in the parent parameter at which point the child will be reset to its starting value. For example,

```
<parameter name="numAgents" type="number" start="1000" end="2000" step="10">
    <parameter name="wealth" type="list" value_type="float" values="10.1 12.3 1234.23 23" />
</parameter>
```

In this case, the sweeper will do set *numAgents* to 1000 and then sweep through each of the the *wealth* values. Then it will increment *numAgents* to 1010, and sweep through all the *wealth* values. It will continue in this fashion until *numAgents* reaches its end value. In this way, the simulation is run such that all the combinations of *numAgents* and *wealth* values are explored.

Random Seed Parameter

The default random seed (i.e. the seed for the default random stream) can be specified in a parameter file using a number type parameter with a name of "randomSeed". For example,

```
<parameter name="randomSeed" type="number" start="1" end="4" step="1"/>
```

If no such parameter exists, the batch mechanism will set the random seed to the current time in milliseconds. **Note that this seed will then be used for all the batch runs, that is, each batch run will use the same random seed.**

2. BSF (Bean Shell Framework) Scripting

A parameter sweep can also be defined programmatically in any script file that can be executed by the bean shell framework. A '**builder**' variable is made available to the script and the script will use this **builder** to define the sweep. The javadoc for the **builder** is available here. An example follows:

```
builder.addConstant("long_const", 11L);
builder.addConstant("string_const", "hello cormac");
builder.addConstant("boolean_const", false);

builder.addStepper("long_param", 1L, 3L, 1L);
builder.addStepper("float_param", .8f, 1.0f, .1f);
builder.addListSetter("string_param", new String[]{"foo", "bar"});
builder.addListSetter("randomSeed", new int[]{1, 2});
```

The top part of this script file adds three constants. The second part creates a chain of parameter setters. A builder has two types of add* methods. The first specifies the parent parameter setter and the second doesn't. When the parent parameter setter is ommitted then the current setter is added as a child of the most recently added setter. In the above, the "float_param" is added as a child of the "long_param", the "string_param" is added as a child of the "float_param" and so on. In this way a chain of setters can be created. The equivalent xml file looks like:

```
<?xml version="1.0"?>
<sweep runs="2">
        <parameter name="long_const" type="constant" constant_type="number" value="11L"/>
        <parameter name="string_const" type="constant" constant_type="string" value="hello
cormac"/>
        <parameter name="boolean_const" type="constant" constant_type="boolean" value="false"/>

        <parameter name="long_param" type="number" start="1L" end="3" step="1">
                <parameter name="float_param" type="number" start=".8f" end="1.0f" step=".1f">
                        <parameter name="string_param" type="list" value_type="String"
values="foo bar">
                                <parameter name="randomSeed" type="list" value_type="int"
values="1 2"/>
                        </parameter>
                </parameter>
        </parameter>
</sweep>
```

For more info about the various builder methods see, the javadocs

3. Groovy scripting

A parameter sweep can also be defined using a groovy script. This takes advantage of groovy's builders concepts. A sample groovy script is given below.

```
builder.sweep(runs : 2) {
  constant(name : 'long_const', value : 11L)
  constant(name : 'string_const', value : "hello cormac")
  constant(name : 'boolean_const', value : false)

  number(name : 'long_param', start : 1L, end : 3, step : 1) {
    number(name : 'float_param', start : 0.8f, end : 1.0f, step : 0.1f) {
      list(name : 'string_param', values : ['foo', 'bar']) {
        list(name : 'randomSeed', values : [1, 2])
      }
    }
  }
}
```

A 'builder' variable is passed into the scripts execution context and the parameter sweep is built using that. Each node in the builder takes a groovy map (e.g. name : 'foo') that constitutes the attributes of that parameter node. Curly brackets "{}" are used to represent nested parameters. The above script defines 3 constants and then a chain of parameters. This script is equivalent to the following xml:

```
<?xml version="1.0"?>
<sweep runs="2">
        <parameter name="long_const" type="constant" constant_type="number" value="11L"/>
        <parameter name="string_const" type="constant" constant_type="string" value="hello 
cormac"/>
        <parameter name="boolean_const" type="constant" constant_type="boolean" value="false"/>

        <parameter name="long_param" type="number" start="1L" end="3" step="1">
                <parameter name="float_param" type="number" start=".8f" end="1.0f" step=".1f">
                        <parameter name="string_param" type="list" value_type="String" 
values="foo bar">
                                <parameter name="randomSeed" type="list" value_type="int" 
values="1 2"/>
                        </parameter>
                </parameter>
        </parameter>
</sweep>
```

The builder node names are explained below.

- sweep - the root node. It takes a single attribute 'runs' which defines the number of times the sweeper will sweep through the defined space in its entirety.
- constant - defines a constant value.
    - name - the name of the parameter. The current value of the parameter can be retrieved from the Parameters object using this name.
    - value - defines the value of the constant. The actual value can be a numeric, boolean or string value.
- number - defines a parameter space with a starting, ending and step value. The sweep mechanism will sweep through the space defined by these values by incrementing the starting value with the step value until the ending value is reached.
    - name - the name of the parameter. The current value of the parameter can be retrieved from the Parameters object using this name.
    - start - the starting numeric value
    - end - the ending numeric value
    - step - the amount to increment
- list - defines a parameter space as a list.
    - name - the name of the parameter. The current value of the parameter can be retrieved from the Parameters object using this name.
    - values - the elements of the list. The value of this attribute must an actual groovy list.

An additional parameter node type 'custom' allows for the programmatic creation of a parameter space. This space is dynamic in that it can change in response to other parameter values or output from the model captured in parameters.

For example, what follows is the same as the above except that the first list parameter node has been replaced by a custom node.

```
import repast.parameter.groovy.*
import repast.parameter.*

builder.root(runs : 2) {
  constant(name : 'long_const', value : 11L)
  constant(name : 'string_const', value : "hello cormac")
  constant(name : 'boolean_const', value : false)

  number(name : 'long_param', start : 1L, end : 3, step : 1) {
    number(name : 'float_param', start : 0.8f, end : 1.0f, step : 0.1f) {
      custom(setter : new StringListSetter()) {
        list(name : 'randomSeed', values : [1, 2])
      }
    }
  }
}

class StringListSetter extends CustomParameterSetter {

  def list = ['foo', 'bar']
  int count = 0

  public StringListSetter() {
    super('string_param', String.class)
  }

  void reset(Parameters params) {
    count = 0
    params.setValue(name, list[count])
    count++
  }

  boolean atEnd() {
    return count == 2
  }

  void next(Parameters params) {
    params.setValue(name, list[count])
    count++
    println "RunState Batch Number: ${runState.runInfo.batchNumber}"
    println "RunState Run Count: ${runState.runInfo.runNumber}"
  }

  void addParameters(ParametersCreator creator) {
    creator.addParameter(name, parameterType, list[count], false)
  }
}
```

The custom node parameter takes a single attribute 'setter'. The value of the setter attribute must be a class that extends CustomParameterSetter. See the javadocs for CustomParameterSetter for more info. Each custom setter has access to the RunState for the next run and can use that in setting parameter's values. In addition, the a custom parameter setter has the opportunity in the addParameters(ParametersCreator) method to add extra parameters beyond the one or ones that it is strictly responsible. In this way, a custom setter could add some parameters that capture some output of a model and set a parameter based on that output. **Note** that the reset method is called to set the values for the first run. Subsequent runs then call the next(Parameters) method to set the next parameter value.

Batch Runs

This page last changed on Aug 12, 2009 by etatara.

Batch Runs

A batch run is can be started from the command line by running the repast.batch.BatchMain class. For example,

```
java -cp ... repast.batch.BatchMain [options] target
```

The target is either the path to a scenario or the fully qualified name of a class that implements BatchScenarioCreator. The options are:

- -help displays usage help
- -interactive specifies that the batch mode scheduler can be programmatically paused or otherwise manipulated
- -params <file> where <file> is the path to a parameter sweep file. More information on parameter sweep files can be found in Batch Parameters
- -opt <file> where file is the path to properties file specifying the properties used to perform the optimizing batch run

The properties file for an optimizing run has the following properties:

- run_result_producer which specifies the fully qualified name of a class that implements the RunResultProducer interface.
- advancement_chooser which specifies the fully qualifed name of a class that implmements the AdvancementChooser interface.
- parameter_file which specifies the path to a parameter sweep file
- bean_shell_script which specifies the path to a bean shell script that defines a parameter sweep.

Of these properties, the run_result_producer must be present as must either a parameter_file or bean_shell_script. If both a parameter_file and bean_shell_script are present then the parameter_file will be used. The advancement_chooser is optional. For more on optimizing batch runs, see ???.

Context

This page last changed on Nov 27, 2007 by etatara.

The Context is the core concept and object in Repast Simphony. It provides a data structure to organize your agents from both a modelling perspective as well as a software perspective. Fundamentally, a context is just a bucket full of agents, but they provide more richness.

Repast S contexts are hierarchically nested named containers that hold model components. The model components can be any type of POJO, including other contexts, but are often expected to be agent objects. Each model component can be present in as many contexts as the modeler desires. The hierarchical nesting means that a model component that is present in a context is also present in all of that context's parent contexts. Of course, the converse is *not* true in the general case. The hierarchical nesting structure itself can be declaratively or imperatively specified by the modeler. Context membership and structure is completely dynamic and agents can be in any number or combination of contexts at any time. Furthermore, agents can themselves contain any number of contexts and even be contexts. In addition, the contents of components within contexts (e.g., agent properties) can be declaratively logged at runtime.

In addition to supporting hierarchical nesting, contexts support *projections*. Repast S projections are named sets of relationships defined over the members of a context. For example, a Repast S network projection stores a network or graph relationship between the members of its context. The members of this context can then ask who they are linked to and who is linked to them. Similarly, the Repast S grid projection stores a set of Cartesian coordinates for each member of the context. The members of this context can ask where they are. For more details on projections, see Details on Projections. Each context can support any mixture of projections. Also, projections can be declaratively visualized at runtime.

Contexts also support *data layers*. Data layers represent numerical data that can be accessed using a set of coordinates. Data layers allow model designers to provide numerical values with which their agents can interact. The data layers for a given context are separate from the layers for other contexts, allowing the agents to orient themselves with respect to a set of data depending on which context they are using. Within a context, data layers can be connected to:

- a particular projection (e.g., each cell in a two-dimensional grid may have some data associated with it),
- multiple projections (e.g., a value at a particular set of coordinates might depend on both a grid and a set of network relations), or
- no projections (e.g., the data is stored in an abstract matrix).

Projections

This page last changed on May 12, 2008 by collier.

While Contexts create a bucket to hold your agents, Projections impose a structure upon those agents. Simply using Contexts, you could never write a model that provided more than a simple "soup" for the agents. The only way to grab other agents would be randomly. Projections allow you to create a structure that defines relationships, whether they be spatial, network, or something else. A projection os attached to a particular Context and applies to all of the agents in that Context. This raises an important point. An object must exist in a Context before it can be used in a projection. So, if you try to work with an agent in a projection before you have added it to the Context, it will fail. For example, if you have a network, and you try to create a link between agents where at least one of the agents does not yet exist in the context, then the operation will fail. This is to maintain the integrity of the context which is used in many place throughout the repast system.

You can apply as many projections to a context as you want, so you could have a context that contains, a grid, a gis space and 4 networks if you felt so inclined. This provides a great deal of flexibility to design your model.

Your agents can get a reference to various projections through their enclosing context. An agent could do the following where "this" refers to the agent:

```
Context context = ContextUtils.getContext (this)

Projection projection = context.getProjection("friendshipNetwork");
```

That projection wouldn't be that useful unless you cast the projection to something more specific. You could also do it like this:

```
Context context = ContextUtils.getContext(this);

Network network = context.getProjection(Network.class, "friendshipNetwork");
```

That would take care of the casting for you.

Instantiating projections

In general, projections are created using a factory mechanism in the following way.

1. Find the factory
2. Use the factory to create the projection

Note that network projections should not longer be created in this way. The factory finder and factory methods will continue to work, but a NetworkBuilder should be used instead. See the Network projection page for details.

For example,

```
Context<SimpleHappyAgent> context = Contexts.createContext(SimpleHappyAgent.class, "my
context");
GridFactory factory = GridFactoryFinder.createGridFactory(new HashMap());
Grid grid = factory.createGrid("Simple Grid", context, ...);
```

```
Context<SimpleHappyAgent> context = Contexts.createContext(SimpleHappyAgent.class, "my
context");
ContinuousSpaceFactory factory = ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(new
HashMap());
ContinuousSpace space = factory.createContinuousSpace("Simple Space", context, ...);
```

Each factory creates a projection of a specific type and requires the context that the projection is associated with and the projections name as well as additional arguments particular to the projection type. These additional arguments are marked above with "..." and are explicated on the individual pages for that projection.

Continuous Space Projection

This page last changed on Aug 10, 2010 by etatara.

A continuous (real valued) space projection is a space in which an agents location is represented by floating point coordinates. For example, one agent may be at the point 2.133, 0 and another may be at 21.1, 234. This contrasts with the discrete integer valued grid projections. However, much of the behavior of a continuous space like that of the grid, so it maybe worthwhile reading those Grid Projections pages first.

Repast Simphony's Continous Space support

Repast Simphony provides a rich set of tools to work with grids of all kinds. Specifically, it supports

- 1D, 2D, 3D and ND continuous spaces,
- Toroidal, Reflective and Fixed Border conditions.

Creating A Continuous Space

Creating a continuous space in Repast Simphony is similar to creating the other projections. Your agents must exist in the context in order to exist in the continuous space, and you want to create the continuous spaceusing a factory to make sure that it is initialized properly.

```
ContinuousSpace factory = ContinuousSpaceFactoryFinder.createGridFactory(new HashMap());
ContinuousSpace space = factory.createContinuousSpace("My Space", context, continuousAdder,
pointTranslator, 10, 10);
```

So, most of that is pretty straight forward, eh? Pass the method a name (so that you can look up your grid later) and the context that will back the continuous space. But what of these additional arguments. Like the Grid, an *Adder* must be specified as well as a *PointTranslator* and the dimensions of the grid.

Adders

Like most projections, an object doesn't exist in the projection until it is added to the context, and once it is added to the context, it is automatically added to the projection. But where do we put it in the continuous space? That's where the *ContinuousSpaceAdder* interface comes in. The adder allows you to specify where to place an object once it has been added to the context. It has a simple method:

```
public void add(ContinuousSpace destination, U object);
```

More generics fun, eh? Well, it's not so bad. In this case, the destination is the projection to which we are adding our object (ContinuousSpace). Object is the object we want to add to that projection. So, <U> is whatever we're adding.   Basically, to implement this interface, you just take in an object and do something with it in the supplied projection. Ok, let's get a bit more concrete. Let's say that we have a ContinuousSpace and when an object is added to the continuous space, we want to locate it randomly. Here is the implementation for that:

```
public void add(ContinuousSpace<T> space, T obj) {
        Dimensions dims = space.getDimensions();
        double[] location = new double[dims.size()];
        findLocation(location, dims);
        while (!space.moveTo(obj, location))  {
                findLocation(location, dims);
        }
}

private void findLocation(double[] location, Dimensions dims) {
        for (int i = 0; i < location.length; i++) {
                location[i] = RandomHelper.getDefault().nextDoubleFromTo(0,
dims.getDimension(i) - 1);
        }
}
```

The add method randomly generates locations and puts the objects into that location (assuming the grid allows the object to be there). This little bit of code sets up a random space for you as you add agents. Not surprisingly, this is the code for the RandomCartesianAdder which comes with Repast Simphony. This behaviour may not be what you want. The most basic Adder we provide is the SimpleCartesianAdder. This Adder doesn't actually locate the object anywhere in the grid. It just allows the object to be used in the grid. The object won't appear to have a location until you call

```
continuousSpace.moveTo(object, x, y);
```

This is a very useful adder because you don't have to know where the agent is supposed to be located in advance.

Let's look at one example of writing your own Adder. In this case, let's assume that our agents have their continuous space coordinates stored. The adder just needs to use those coordinates to locate the agents properly. We'll call the Interface Located:

```
public interface Located{

        public int getX();

        public int getY();

}
```

Pretty simple, eh? Let's write an adder that know how to work with this:

```
public class LocatedAdder<Located> implements ContinuousAdder<Located> {

        public void add(ContinuousSpace<Located> grid, Located object){
                grid.moveTo(object, object.getX(), object.getY());
        }

}
```

This will add the object to the grid at the location specified by the agent itself. This doesn't handle any errors or anything, but it should give you an idea of how to create your own adder.

PointTranslators

Just like the *GridPointTranlator* described on the Grid's page, a *PointTranslator* determines the border behavior of a continuous space. The border behavior is what happens when an agents moves past the border of a continuous space. Five kinds of border behavior classes are described below together with a description of how they behave in response to a continuous space's *moveTo* and *moveBy** (the *moveBy** methods are *moveByDisplacement* and *moveByVector*) methods.

1. StrictBorders -- defines a strict boundary that cannot be crossed without throwing a SpatialException. Consequently, any moveTo or moveBy* across the border will throw an exception.
2. StickyBorders -- defines a "sticky" border that to which an agent will "stick" in a moveBy*. However, a moveTo across the border will still throw a SpatialException.
3. InfiniteBorders -- defines infinite borders that the agent will never cross.
4. BouncyBorders -- defines a bouncy border that the agent will bounce off at the appropriate angle in a moveBy*. However, a moveTo across the border will still throw a SpatialException.
5. WrapAroundBorders -- defines borders that wrap around to their opposite borders. This mean moveTo and moveBy* across the border will cross the border and enter the opposite side.

In general then, moveTo will throw a SpatialException when crossing the border and moveBy* will invoke the border behavior.

Moving in a Continuous Space

Movement on a tcontinuous space is accomplished with 3 methods.

- moveTo(T object, double... newLocation)

This specifies the object to move and the new location to move to. It will return true if the move succeeds, for example, if the new location doesn't violate any border behaviors. This method can be used to introduce objects into the grid if they do not yet have a grid location. This will throw a SpatialException if the object is not already in the space, if the number of dimensions in the location does not agree with the number in the space, or if the object is moved outside the continuous space's dimensions.

- NdPoint moveByDisplacement(T object, double... displacement)

Moves the specified object from its current location by the specified amount. For example, moveByDisplacement(object, 3, -2, 1) will move the object by 3 along the x-axis, -2 along the y and 1 along the z. The displacement argument can be less than the number of dimensions in the space in which case the remaining arguments will be set to 0. For example, moveByDisplacement(object, 3) will move the object 3 along the x-axis and 0 along the y and z axes, assuming a 3D continuous space. This will return the new location as a NdPoint if the move was successful, otherwise it will return null. Like moveTo it will throw a SpatialException if the object is not already in the space or if the number of dimensions in the displacement is greater than the number of continuous space dimensions.

- NdPoint moveByVector(T object, double distance, double... anglesInRadians)

Moves the specifed object the specified distance from its current position along the specified angle. For example, moveByVector(object, 1, Direction.NORTH) will move the object 1 unit "north" up the y-axis, assuming a 2D continuous space. Similarly, grid.moveByVector(object, 2, 0, Math.toRadians(90), 0) will rotate 90 egrees around the y-axis, thus moving the object 2 units along the z-axis. Note that the radians / degrees are incremented in a anti-clockwise fashion, such that 0 degrees is "east", 90 degrees is "north", 180 is "west" and 270 is "south." This will return the new location as a NdPoint if the move was successful, otherwise it will return null. Like moveTo it will throw a SpatialException if the object is not already in the space or if the number of dimensions in the displacement is greater than the number of continuous space dimensions.

GIS Projections

This page last changed on Nov 28, 2007 by collier.

Repast Simphony's GIS support is based on the Geography projection. A Geography is essentially a space in which agents are associated with a spatial geometry (polygon, point, etc.. Agents are stored by type in layers within the Geography and agents of the same type must have the same geometry. The geography can be queried spatially to return all the agents within some particular area. When a geography is displayed the agents are displayed in typical GIS fashion and represented by their associated geometries.

Repast Simphony uses Geotoolsand the Java Topology Suiteextensively for its GIS support.

Creating a Geography

Creating a geography in repast simphony is similar to creating the other projections. Your agents must exist in the context in order to exist in the geography, and you want to create the geograhy using a factory to make sure that it is initialized properly.

```
GeographyParameters<Object> params = new GeographyParameters<Object>();
GeographyFactory factory = GeographyFactoryFinder.createGeographyFactory(null);
Geography geography = factory.createGeography("GeograhpyName", context, params);
```

This should be fairly straightforward. We create a GeographyFactory and the create the Geography with that, passing the name of the Geography, the associated context, and the GeographyParameters.

GeographyParameters

GeographyParameters can be used to set the "adder" and the CoordinateReferenceSystem for a Geography.

Adders

The adder controls how any agent added to the context is added to the Geography. The default is a SimpleAdder which although it makes the Geography aware of the agent, it does not move the agent into the Geography by associating it with some Geometry. Custom adders can be created by implementing the GISAdder interface. A typical custom adder would call *Geography.move* to move an agent into the Geography and associate it with a Geometry in some model specific manner.

CoordinateReferenceSystem

Using GeographyParameters, you can define the CoordinateReferenceSystem for the created Geography. In the absence of any user supplied CRS, the default is WGS84 as defined by Geotools' DefaultGeographicCRS.WGS84. **That CRS uses (longitude, latitude) coordinates with longitude values increasing East and latitude values increasing North. Angular units are degrees and prime meridian is Greenwich.**

Moving in a Geography

Moving a Geography is typically accomplished using the following method

```
move(T agent, Geometry geom)
```

The first time this is called for an agent that agent is moved into the Geography and associated with the specific geometry. The specific geometry establishes the agent's location in the geographical space and subsequent queries of the Geography will make use of that Geometry. If this is the first time an agent of this Class has been move in the Geography that type of agent will become associated with that Geometry type. For example, if the agent's type is the Java class anl.model.Truck, and the Geometry is a Point, then all Trucks are considered to be Points and any Truck moved in the Geography must have a Point geometry.

An agent's current Geometry (i.e. its location) can be retrieved with:

```
Geometry getGeometry(Object agent);
```

This returned Geometry can be used to move the agent without recreating a Geometry and passing that new Geometry to the move method. For example, where "this" refers to an agent,

```
Geometry geom = geography.getGeometry(this);
Coordinate coord = geom.getCoordinate();
coord.x += .005;
coord.y += .005;
geography.move(this, geom);
```

In this case, we know the Geometry is a Point and so it has a single coordinate whose x and y values we can change. Note that its important to call move here to register the move with the Geography.

Geography has additional move methods that can move by displacement etc.:

```
/**
   * Displaces the specified object by the specified lon and lat amount.
   *
   * @param object the object to move
   * @param lonShift the amount to move longitudinaly
   * @param latShift the amount to move latitudinaly
   * @return the new geometry of the object
   */
  Geometry moveByDisplacement(T object, double lonShift, double latShift);
```

```
/**
   * Moves the specified object the specified distance along the specified angle.
   *
   * @param object the object to move
   * @param distance the distance to move
   * @param unit the distance units. This must be convertable to meters
   * @param angleInRadians the angle along which to move
   *
   * @return the geometric location the object was moved to
   */
  Geometry moveByVector(T object, double distance, Unit unit,  double angleInRadians);
```

```
/**
   * Moves the specified object the specified distance along the specified angle.
   *
   * @param object the object to move
   * @param distance the distance to move in meters
   * @param angleInRadians the angle along which to move
   *
   * @return the geometric location the object was moved to
   */
  Geometry moveByVector(T object, double distance,  double angleInRadians);
```

Queries

A Geography can also be queried spatially to return the objects within a particular envelope.

```
 /**
   * Gets an iterable over all the objects within the specified envelope.
   *
   * @param envelope the bounding envelope
   * @return an iterable over all the objects within the specified location.
   */
  Iterable<T> getObjectsWithin(Envelope envelope);
```

Additional queries are available for getting agents that intersect another agent or geometry, for getting the agents that touch another agent or geometry, and so on. See the classes in the repast.simphony.query.space.gis package for more information.

Grid Projections

This page last changed on Aug 10, 2010 by etatara.

Grids are a mainstay of agent based simulation. Many of the earliest simulation such as Schelling's Dynamic Models of Segregation, heatbugs, and sugarscape were originally built on grids. Basically a grid, for our purposes, are a 1 or more dimensional data structure that is divided up into a number of cells. These cells can be referenced by their integer coordinates. In other words, a grid is an n-dimensional matrix. Even though grids have been used in agent based simulation since it's inception (and before), they still have value today. These data structures are very efficient and provide a number of different ways to define neighborhoods. While they don't offer the flexibility of an abstract network or the realism of a continuous or GIS space, the grid can be used to simulate spaces and to create highly structured relationships between agents. While most people think of grids to represent space, there is no reason why a grid couldn't be used define more abstract kinds of relationships. No matter how you look at it, grids are still a powerful tool in an overall agent based toolkit.

Repast Simphony's Grid support

Repast Simphony provides a rich set of tools to work with grids of all kinds. Specifically, it supports

- 2D, 3D and ND grids,
- querying for Von Neumann and Moore neighborhoods,
- Toroidal, Reflective and Fixed Border conditions.

Creating A Grid

Creating a grid in Repast Simphony is similar to creating the other projections. Your agents must exist in the context in order to exist in the grid, and you want to create the grid using a factory to make sure that it is initialized properly.

```
GridFactory factory = GridFactoryFinder.createGridFactory(new HashMap());
Grid grid = factory.createGrid("My Grid", context, gridBuilderParameters);
```

So, most of that is pretty straight forward, eh? Pass the method a name (so that you can look up your grid later) and the context that will back the grid. But what is that last argument? The gridBuilderParameters.   GridBuilderParameters are used to specify the properties of the grid itself, such as its dimensions, its border behavior, and how agents are initially added to the grid.

GridBuilderParameters

The GridBuilderParameters class contains a variety of static methods for creating specific types of GridBuilderParameters. For example,

```
GridBuilderParameters params = GridBuilderParameters.singleOccupancy2DTorus(new
SimpleAdder<Agent>(), 10, 10);
```

will create parameters for a 10x10 torus where each cell can only have a single occupant. Other static methods (see the javadocs) allow you to create other types of grids. In general though, youwill always have to specify a *GridAdder* and perhaps a *GridPointTranslator* when creating GridBuilderParameters. You will also have to specify the dimensions. These will be the last arguments to the method.

Adders

Like most projections, an object doesn't exist in the projection until it is added to the context, and once it is added to the context, it is automatically added to the projection. But where do we put it in the grid? That's where the *GridAdder* interface comes in. The adder allows you to specify where to place an object once it has been added to the context. It has a simple method:

```
public void add(Grid destination, U object);
```

More generics fun, eh? Well, it's not so bad. In this case, the destination is the type of projection to which we are adding our object (a grid). Object is the object we want to add to that projection. So, <U> is whatever we're adding.   Basically, to implement this interface, you just take in an object and do something with it in the supplied projection. Ok, let's get a bit more concrete. Let's say that we have a Grid and when an object is added to the grid, we want to locate it randomly. Here is the implementation for that:

```
public void add(Grid<T> space, T obj) {
        Dimensions dims = space.getDimensions();
        int[] location = new int[dims.size()];
        findLocation(location, dims);
        while (!space.moveTo(obj, location))  {
                findLocation(location, dims);
        }
}

private void findLocation(int[]location, Dimensions dims) {
        for (int i = 0; i < location.length; i++) {
                location[i] = RandomHelper.getDefault().nextIntFromTo(0, dims.getDimension(i) -
1);
        }
}
```

The add method randomly generates locations and puts the objects into that location (assuming the grid allows the object to be there). This little bit of code sets up a random space for you as you add agents. Not surprisingly, this is the code for the RandomGridAdder which comes with Repast Simphony. This behaviour may not be what you want. The most basic Adder we provide is the SimpleAdder; This Adder doesn't actually locate the object anywhere in the grid. It just allows the object to be used in the grid. The object won't appear to have a location until you call

```
grid.moveTo(object, x, y);
```

This is a very useful adder because you don't have to know where the agent is supposed to be located in advance.

Let's look at one example of writing your own Adder. In this case, let's assume that our agents have their grid coordinates stored. The adder just needs to use those coordinates to locate the agents properly. We'll call the Interface Located:

```
public interface Located{

        public int getX();

        public int getY();

}
```

Pretty simple, eh? Let's write an adder that know how to work with this:

```
public class LocatedAdder<Located> implements GridAdder<Located> {

        public void add(Grid<Located> grid, Located object){
                grid.moveTo(object, object.getX(), object.getY());
        }

}
```

This will add the object to the grid at the location specified by the agent itself. This doesn't handle any errors or anything, but it should give you an idea of how to create your own adder.

GridPointTranslators

A *GridPointTranslator* determines the border behavior of a grid. The border behavior is what happens when an agents moves past the border of a grid. Five kinds of border behavior classes are described below together with a description of how the behave in response to a Grid's *moveTo* and *moveBy** (the *moveBy** methods are *moveByDisplacement* and *moveByVector*) methods.

1. StrictBorders -- defines a strict boundary that cannot be crossed without throwing a SpatialException. Consequently, any moveTo or moveBy* across the border will throw an exception.
2. StickyBorders -- defines a "sticky" border that to which an agent will "stick" in a moveBy*. However, a moveTo across the border will still throw a SpatialException.
3. InfiniteBorders -- defines infinite borders that the agent will never cross.
4. BouncyBorders -- defines a bouncy border that the agent will bounce off at the appropriate angle in a moveBy*. However, a moveTo across the border will still throw a SpatialException.
5. WrapAroundBorders -- defines borders that wrap around to their opposite borders. This mean moveTo and moveBy* across the border will cross the border and enter the opposite side. A 2D grid with WrapAroundBorders is a torus.

In general then, moveTo will throw a SpatialException when crossing the border and moveBy* will invoke the border behavior.

Moving on a Grid

Movement on the grid is accomplished with 3 methods.

- moveTo(T object, int... newLocation)

This specifies the object to move and the new location to move to. It will return true if the move succeeds, for example, if the location is unoccupied an single occupancy grid and the new location doesn't violate any border behaviors. This method can be used to introduce objects into the grid if they do not yet have a grid location. This will throw a SpatialException if the object is not already in the space, if the number of dimensions in the location does not agree with the number in the space, or if the object is moved outside the grid dimensions.

- GridPoint moveByDisplacement(T object, int... displacement)

Moves the specified object from its current location by the specifiedamount. For example, moveByDisplacement(object, 3, -2, 1) will move the object by 3 along the x-axis, -2 along the y and 1 alongthe z. The displacement argument can be less than the number of dimensions in the space in which case the remaining argument will beset to 0. For example, moveByDisplacement(object, 3) will move the object 3 along the x-axis and 0 along the y and z axes, assuming a 3D grid. This will return the new location as a GridPoint if the move was successful, otherwise it will return null. Like moveTo it will throw a SpatialException if the object is not already in the space or if the number of dimensions in the displacement is greater than the number of grid dimensions.

- GridPoint moveByVector(T object, double distance, double... anglesInRadians)

Moves the specifed object the specified distance from its current position along the specified angle. For example, moveByVector(object, 1, Direction.NORTH) will move the object 1 unit "north" up the y-axis, assuming a 2D grid. Similarly, grid.moveByVector(object, 2, 0, Math.toRadians(90), 0) will rotate 90 egrees around the y-axis, thus moving the object 2 units along the z-axis. Note that the radians / degrees are incremented in a anti-clockwise fashion, such that 0 degrees is "east", 90 degrees is "north", 180 is "west" and 270 is "south." This will return the new location as a GridPoint if the move was successful, otherwise it will return null. Like moveTo it will throw a SpatialException if the object is not already in the space or if the number of dimensions in the displacement is greater than the number of grid dimensions.

Network Projections

This page last changed on Aug 10, 2010 by etatara.

Many models use abstract relationships between agents using Networks or Graphs. These connections may represent social connections, physical infrastructure connections or some other abstract connections. Repast provides tools to work with Networks or Graphs easily as a Projection. The default support for networks is supplied by the excellent library JUNG.

Building a Network

In many situations, you won't need to build the network yourself.   If you are using a persistent Data Loader to load the network from a freezedryed instance the networks will be generated for you. However, if you using your own ContextBuilder or you wish to create networks during your simulation, you will need to be able to build networks in code.

The first thing to remember when working with networks is that, like most projections, in order to participate in a network, and agent must already exist in the context. A side affect of this is that any agent in a context is automatically a member of any networks associated with that context. The agent might not have any edges, but they will be in the network for all intents and purposes.

Network Builders

Networks are created using a NetworkBuilder. The NetworkBuilder class allows the user to tune the network creation to create specific kinds of networks. This "tuning" is done by calling various methods on the builder. If none of these methods are called then a simple "plain vanilla" network is created. For example,

```
NetworkBuilder builder = new NetworkBuilder("Network", context, true);
Network network = builder.buildNetwork()
```

The first argument to the builder constructor is the name of the network, the second is the context which the network "projects" and the last is whether or not the network is directed. By default, the *addEdge(source, target, ...)* methods of a network will create and use an instance of RepastEdge as the edge between the source and target. Using the builder, you can customize this edge creation process by specifying an EdgeCreator on the NetworkBuilder. The EdgeCreator interface looks like:

```
/**
 * Factory class for creating edges.
 *
 * @author Nick Collier
 */
public interface EdgeCreator<E extends RepastEdge, T> {

  /**
   * Gets the edge type produced by this EdgeCreator.
   *
   * @return the edge type produced by this EdgeCreator.
   */
  Class getEdgeType();
  /**
   * Creates an Edge with the specified source, target, direction and weight.
   *
   * @param source the edge source
   * @param target the edge target
   * @param isDirected whether or not the edge is directed
   * @param weight the weight of the edge
   * @return the created edge.
   */
  E createEdge(T source, T target, boolean isDirected, double weight);

}
```

The intention is that the user implement the createEdge method to customize the edge creation process. All edges created on the network via *addEdge(source, target, ...)* will forward edge creation to this method. Specifying the edge creator on the builder looks like:

```
NetworkBuilder builder = new NetworkBuilder("Network", context, true);
builder.setEdgeCreator(myEdgeCreator);
Network network = builder.buildNetwork()
```

Any networks created using this builder will then use that EdgeCreator.

Networks can also be loaded from a file. Currently, the builder can load networks from UCINet's dl format or from Excel. The relevant methods in the NetworkBuilder are:

```
/**
   * Sets this NetworkBuilder to create the network from a file. This
   * will use the first network found in the file.
   *
   * @param fileName    the name of the file to load from
   * @param format      the format of the file
   * @param nodeCreator a node creator that will be used to create the agents / nodes
   * @return this NetworkBuilder
   * @throws IOException if there is a file related error
   */
  public NetworkBuilder load(String fileName, NetworkFileFormat format, NodeCreator
nodeCreator) throws IOException
```

```
/**
   * Sets this NetworkBuilder to create the network from a file. This
   * will use the first network found in the file.
   *
   * @param fileName    the name of the file to load from
   * @param format      the format of the file
   * @param nodeCreator a node creator that will be used to create the agents / nodes
   * @param matrixIndex the index of the matrix in the file to load. Starts with 0.
   * @return this NetworkBuilder
   * @throws IOException if there is a file related error
   */
  public NetworkBuilder load(String fileName, NetworkFileFormat format, NodeCreator
nodeCreator, int matrixIndex) throws IOException
```

The NodeCreator in the above is used to perform the actual node / agent creation based on the information in the network file. These nodes are added to the context and edges between them are created using the link data from the file. The NetworkFileFormat is either NetworkFileFormat.DL or NetworkFileFormat.EXCEL, although more formats may be added in the future. The Excel file format is described below.

```
The matrix is assumed to be square.
```

```
Each worksheet is treated as a matrix, and any worksheets
that do not contain matrices will cause an error. The worksheet name
is treated as the matrix label unless the name begins with Sheet
(Excel's generic worksheet name). The format for excel files is that
imported and exported by UCINet. The first cell is empty, and the
node labels begin on this first row in the second column. The column
node labels begin in first column on the second row. The actual data
begins in cell 2,2. For example,

              | first_label | second_label | ...
-----------+-----------+-------------+----
first_label  | 0           | 1               | ...
-----------+----------+--------------+----
second_label | 1           | 0            | ...
 -------------+-------------+-------------+----
 ...          | ...         | ...          | ...

If the matrix has no node labels, repast will expect the first row and
column to be blank and as before, for the data to begin in cell 2,2.<p>
```

An example of the network builder loading a file:

```
NetworkBuilder builder = new NetworkBuilder("Network", context, true);
builder.load("./repast.simphony.core/test/repast/simphony/graph/matrices.xls",
NetworkFileFormat.EXCEL, myNodeCreator);
Network net = builder.buildNetwork();
```

Lastly, a NetworkGenerator can be set on the NetworkBuilder. NetworkGenerators typically assume that the context already contains the nodes / agents and the generator arranges them in the appropriate topography (e.g. a Small World). For example,

```
// make sure the agents are in the context before
// creating the network with the generator
for (int i = 0; i < 25; i++) {
      VizAgent agent = new VizAgent("Agent-" + i);
      context.add(agent);
    }

NetworkGenerator gen = new WattsBetaSmallWorldGenerator(.2, 2, false);
NetworkBuilder builder = new NetworkBuilder("Electricity Network", context, true);
builder.setGenerator(gen);
Network net = builder.buildNetwor();
```

The arguments to the generator are the probability of rewiring the network, the local neighborhood size, and whether or not generated edges will be created symmetrically. We then create a NetworkFactory and use that to create the network, passing it the network name, the context, the generator, and whether or not the resulting network will be directed. In this case, the generator creates a small world type network using the agents in the context as nodes.

See the repast.simphony.context.space.graph package for more information on NetworkGenerators and what is available.

Note that although the generators supplied by simphony do assume that the context already contains nodes / agents, the NetworkGenerator interface is flexible and can be used to implement any sort of custom network creation. For example, the network file loaders are actually implements as NetworkGenerators whose constructors take the necessary NodeCreator etc. information.

Working with a network

Once you have a network, you can always add Relationships, but that's not really that useful until you can query based on the relationships. The Network library provides all of the methods you would expect to have to access the objects and relationships in it. The particular methods in which you will be interested are:

- getAdjacent(agent) - Retrieves all of the objects that share a relationship with this agent
- getPredecessors(agent) - Retrieves all of the objects that have a relationship to this agent
- getSuccessors(agent) - Retrieves all of the objects that have a relationship from this agent
- getEdges(agent) - Retrieves all of the edges which have this agent as an endpoint
- getOutEdges(agent) - Retrieves all of the edges which have this agent as a source
- getInEdges(agent) - Retrieves all of the edges which have this agent as a target
- addEdge(agent, agent) - Adds an edge between the two agents
- addEdge(agent, agent, weight) - Adds an edge of the specified weight between the two agents
- addEdge(RepastEdge edge) - Adds the specified edge. Assumes that the source and target of the edge are part of the network
- getEdge(agent, agent) - Get the edge, if any, between source and target agents
- ...

In addition to these, there are also many other methods. See the javadoc for Network for more info.

Context Loading

This page last changed on May 12, 2008 by collier.

Loading a Context

Context creation and loading is where your model is built. In some cases, simphony will automatically create and load a context for you. Is this is typically the case if you use the visual editor to create your agents etc., or you load your model from a freezedryed instance. However, in many other cases, you'd like to control the model creation process and for this you need to implement a ContextBuilder.

```
/**
 * Interface for classes that build a Context by adding projections,
 * agents and so forth.
 *
 */
public interface ContextBuilder<T> {

        /**
         * Builds and returns a context. Building a context consists of filling it with
         * agents, adding projects and so forth. The returned context does not necessarily
         * have to be the passed in context.
         *
         * @param context
         * @return the built context.
         */
        Context build(Context<T> context);
}
```

Once you've implemented such, you can then use the data loader wizard to specify your custom ContextBuilder as the "Specific Java Class" to use to load your model. If you are creating the master context this way, you can just use the Context that simphony passees in to the build method.

Note that if your ContextBuilder implements Context (i.e extends DefaultContext) then Simphony will in fact pass your ContextBuilder to an instance of itself in the build method. Having your ContextBuilder extend DefaultContext is useful in that you can then use @ScheduledMethod on methods in your ContextBuilder to schedule model behaviors at something other than the individual agent level.

Whenever possible, the runtime will try to create the Context and pass it to the ContextBuilder. This will occur under the following situations:

1. model.score file SContext implementation class implements ContextBuilder (**but not** Context). In this case, the runtime will create a DefaultContext and pass that to the ContextBuilder specified in the .score file.
2. model.score file SContext implementation class implements ContextBuilder and Context. In this case, the runtime will pass this class to itself as it is both a Context and a ContextBuilder.
3. model.score file SContext implementation class implements Context (**but not** ContextBuilder). In this case, the runtime will create that Context and pass it to whatever ContextBuilder has been specifed externally (e.g. added through the gui).

The context returned from the ContextBuilder doesn't necessarily have to be the one passed in. In the case of a subcontext, if the passed in context is different from the returned one, the runtime removes the old context and adds this new one as a subcontext to the appropriate parent.

Context and SubContexts

SubContexts can be created, added to parent contexts, and built in the following ways:

1. Do everything in the parent ContextBuilder. So, when the parent context is built, subcontexts are created, added to the parent and loaded with projections, agents, etc. In this case, an SContext Implementation entry for the subcontext in the model.score file is not required.
2. Subcontexts have their own SContext Implementation class that implements ContextBuilder or Context (see above). In this the subcontexts are passed to their own ContextBuilder for building. The actual creation of the subcontext (e.g new DefaultContext("svabhava")) can happen either in the parent context's ContextBuilder or the runtime will create the subcontexts and add them to their parent. In this latter case, for example, if after a context has been created and built, it has no subcontexts but the .score file defines subcontexts for that context then those subcontexts will be created and added to that context. This assumes of course that the subcontexts have their own ContextBuilder defined so that they too can be built.

All this provides flexibility, but it can be confusing. Unless there is a compelling reason not to, its probably easiest to just do everything in the master context ContextBuilder -- creating subcontexts, building them and so forth. This applies to creating ContextBuilders by hand. The situation using codegen will probably be different.

Creating a Dynamic Wizard

This page last changed on Nov 19, 2007 by etatara.

I've made a wizard framework that dynamically loads its options based on "options" specified in plugin.xmls. I've used it in enough places that I've gone ahead and generalized it. If you wish to use this setup I made it fairly straightforward.

The classes used in this are in the repast.simphony.plugin.util plugin and in the repast.simphony.plugin.wizard package. These classes are WizardOption and WizardPluginUtil, DynamicWizard, and some related to the wizard's model (DynamicWizardModel, WizardModelFactory).

WizardOption

WizardOption represents an option that will be taken in a wizard. It contains a SimplePath (the GUI steps the wizard will take) and some description methods (for grabbing the title and description of the option). Generally someone will subclass/subinterface this interface with their own type (say MyWizardOption) and add any extra methods to it.

Implementations of WizardOption (or MyWizardOption) would generally specify their description (to be in tooltips or a description field in the wizard option selection) and a title (for the option's title in a "Choose type" list), and then fill in the other methods as needed.

The implementation would also build the steps it needs (say select file name or select agent class steps) in getWizardPath and return them.

WizardPluginUtil

The only method in WizardPluginUtil is the only one needed to dynamically load up the wizard options. This takes in the plugin id, the extension point id where wizard options would be registered, the specific WizardOption class the options should be an instance of, a PluginManager that the extension points would be grabbed from, and finally a boolean parameter that specifies whether or not to register the options with the DynamicWizard (this is important for using the DynamicWizard, and generally should be true). This assumes the extension point will have a class field specified in it, so it could look like so:

```
<extension-point id="repast.simphony.data.mappingwizard.options">
<parameter-def id="class"/>
</extension-point>
```

Or alternatively, you can just create a new extension point inherting its properties off the default one:

```
<extension-point id="wizard.options" parent-plugin-id="repast.simphony.plugin.util"
                                parent-point-id="wizard.options">
</extension-point>
```

And extensions would then look like (assuming the plugin id is repast.simphony.data)

```
<extension plugin-id="repast.simphony.data"
                point-id="repast.simphony.data.mappingwizard.options"
                id="data.bsf.basicoption">

<parameter id="class" value="repast.data.bsf.gui.BasicScriptOption"/>
</extension>
```

Loading the options

To load these options you have to call the WizardPluginUtil method from some point where you have access to the PluginManager. For now the best way to do this is from a menu or other item action creator. Hopefully at some point a more generic method will exist. You then can store the loaded options somewhere so that you can have direct access to them if you desire. So for instance this could look like

```
public void init(PluginManager manager) {
        // grab all the available wizard options
 List<DataLoaderWizardOption> options = WizardPluginUtil.loadWizardOptions(manager,
                DataLoaderWizardOption.class, PACKAGE_ID, WIZARD_OPTIONS_ID, true);
        // store them for later
 staticOptions = options;
}
```

Creating the wizard

Finally, to actually display a wizard that uses these options you use DynamicWizard. There are a lot of arguments in the DynamicWizard as of now, so it is best to wrap the creation of the wizards in some other static method. So an example method could look like:

```
public static DynamicWizard create(Scenario scenario, Object contextID) {
        return new DynamicWizard(WIZARD_OPTIONS_ID, scenario, contextID, "Select Data Source
Type",
                        "Choose a type for your data source.", new WizardFinishStep(),
                        new WizardModelFactory() {
                                public DynamicWizardModel create(BranchingPath path, Scenario
scenario,
                                                Object contextID) {
                                        return new ChartWizardModel(path, scenario, contextID);
                                }
                        });
}
```

This will create a wizard that has its options stored with the key WIZARD_OPTIONS_ID, scenario and contextID (just general properties), the specified title for the wizard's select an option step, and specified prompt for that same step, a final "wizard has completed" step, and a factory for generating the model used by the wizard (in this case a ChartWizardModel).

Each of these properties plays their own role in the wizard process. Explicitly, the important ones and their function (from this example):

- the wizard id (WIZARD_OPTIONS_ID) - a String that is the extension point id that the wizard options would have been registered under. This in general is just the wizard's ID, meaning, the id the wizard options would have been registered to WizardPluginUtil's methods, but if you set the register flag to true when using loadWizardOptions, it will store the options with the extension point id as the wizard id.

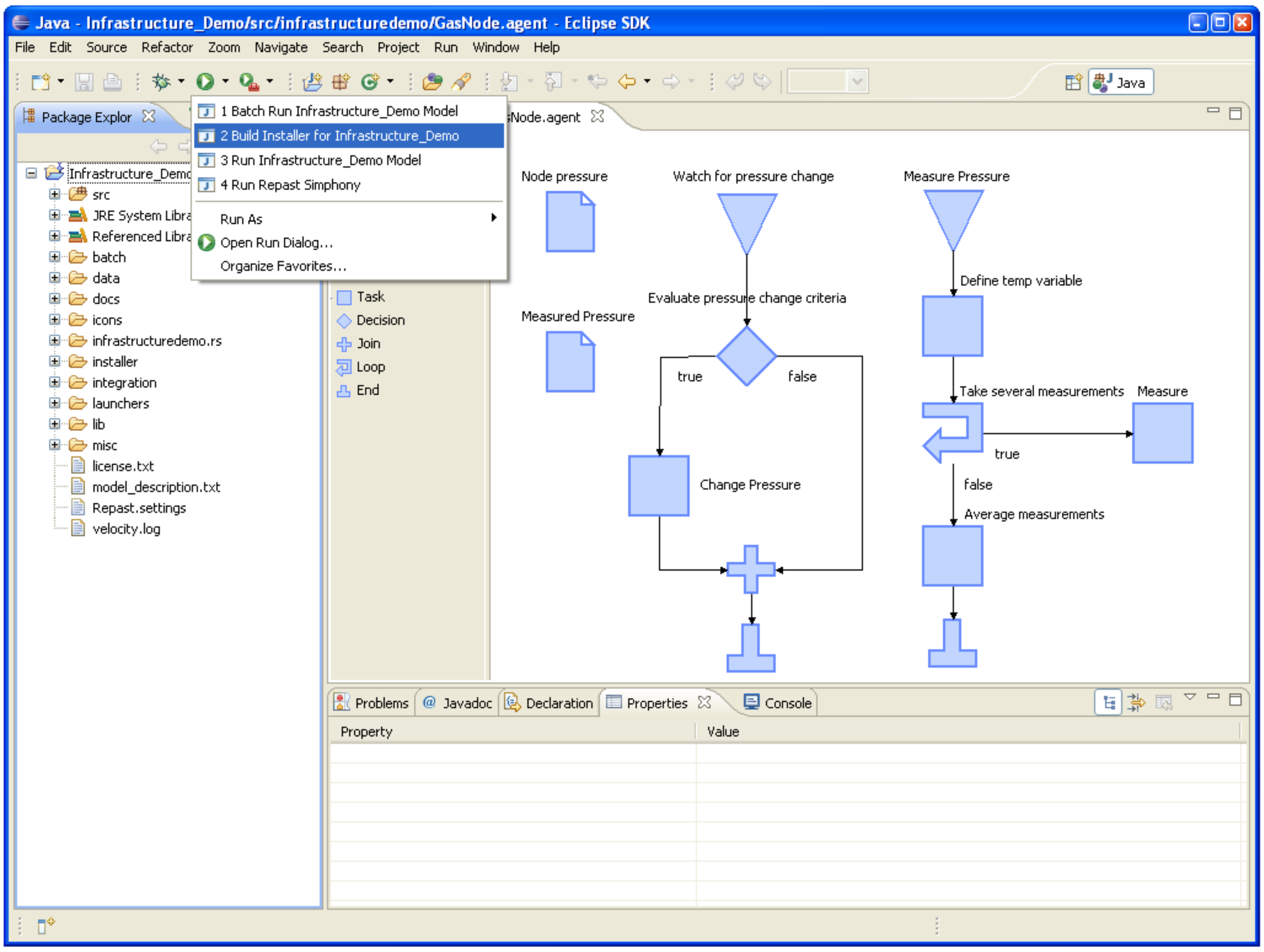
- the wizard model factory - This should build the wizard model that the wizard will pass along to each of the steps. In general this model should be where each step stores their final information. So if one step is in charge of specifying the directory some action will point to, it should when its done (in its applyState method) set this directory on the model. The model itself can be grabbed during the step's init(WizardModel) call, which occurs when the step is displayed. This model is important so that when the wizard has completed you can load the settings that were specified in the wizard steps (as needed).

Distributing your model

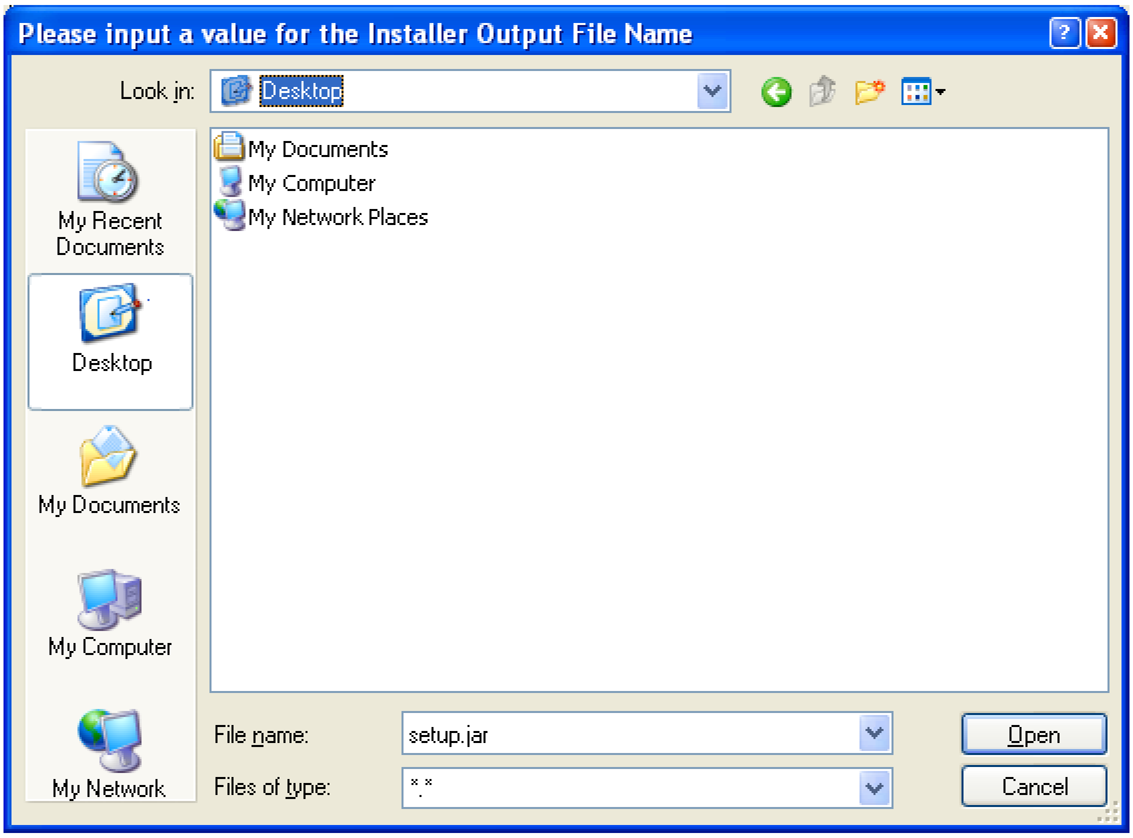
This page last changed on Nov 22, 2010 by north.

Models created with Repast Simphony may easily be distributed with the provided Install Builder tool. This tool builds a distributable installer that anyone may use to install the model to their computer. The only requirement for those to who the model is distributed is the target system to have a current Java Runtime Environment (version 5 or later).

To start the Install Builder, select Build Installer in the run menu



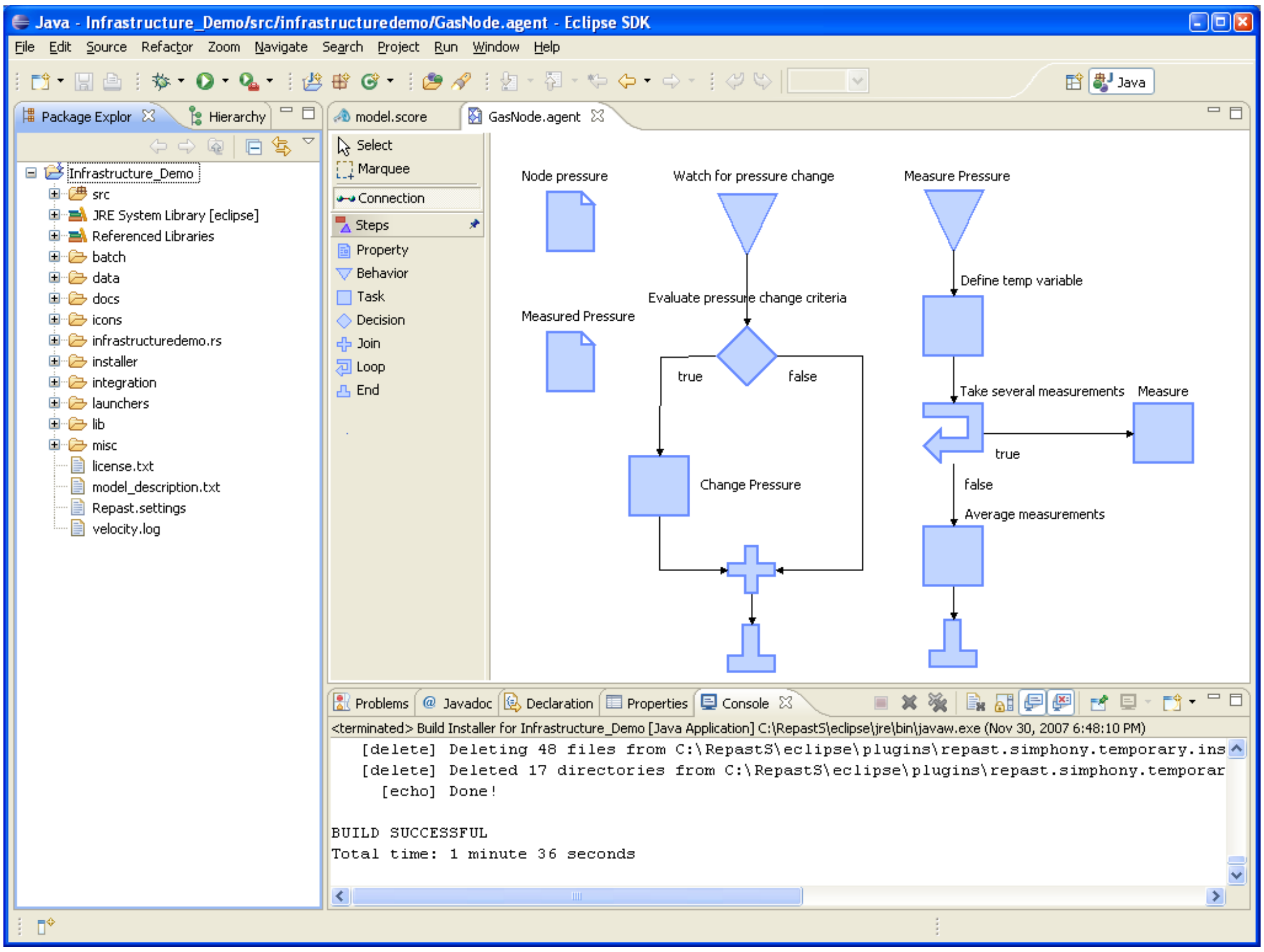
Next enter the location were the installer file will be stored



File name: setup.jar

Files of type: *.*

The builder make take a few minutes to complete the installer. Once this process is complete, a message BUILD SUCCESSFUL should be visible in the Console window.



```
<terminated> Build Installer for Infrastructure_Demo [Java Application] C:\RepastS\eclipse\jre\bin\javaw.exe (Nov 30, 2007 6:48:10 PM)
    [delete] Deleting 48 files from C:\RepastS\eclipse\plugins\repast.simphony.temporary.ins
    [delete] Deleted 17 directories from C:\RepastS\eclipse\plugins\repast.simphony.temporar
      [echo] Done!

BUILD SUCCESSFUL
Total time: 1 minute 36 seconds
```

Enabling FreezeDrying of Custom Projections

This page last changed on Nov 19, 2007 by etatara.

When building a new Projection, some things should be done if it is to be integrated into the whole system. In particular, if it is to be freeze dryable, some special considerations may need to be made. Since the freeze dryer generally just writes out all the fields of an object, this may or may not function properly for a given Projection.

For a projection to be properly freeze dryed one of the following has to happen:

1. The Projection and all the objects it contains should have a default constructor, or all of its constructors should be able to support taking in dummy (or null) values. In general, this means the Projection should be able to be written out and read in with the default freeze dryer (DefaultFreezeDryer).
2. A custom FreezeDryer should be written that will handle the given projection class. This freeze dryer should be able to read and write the default implementation of your projection, and if reasonable, other implementations.
3. A ProjectionDryer should be created. This is not a FreezeDryer, but is an interface used by the Context FreezeDryer. Building one of these is discussed below.

Building a ProjectionDryer

All a ProjectionDryer does is store the settings from a custom Projection into a map, and later rebuild a projection from a map (with the same settings). Implementations store properties in the addProperties method, rebuild the projection in instantiate, and restore properties in the loadProperties methods.

addProperties

The addProperties method should add implementation specific properties to the passed in Map. For instance, in a Network these properties could be a list of edges in the network. This would be something of the form "edges"->((node1, node2, edgeWeight), (node2, node3, 1.2), ...)). So it would be implemented by:

```
protected void addProperties(Context context, MyNetwork net, Map<String, Object> properties) {
        super.addProperties(context, net, properties);
        ArrayList list = new ArrayList();
        for (Edge edge : net.getEdges()) {
                list.add(new Object[] { edge.getSource(), edge.getTarget(), edge.getWeight()
});
        }
        properties.put("edges", list);
}
```

The super call lets the class's superclass (assuming it is not ProjectionDryer) add any neccessary properties. It then builds a list of the edges, storing the properties in an array. Finally it stores to the properties map the list of edges.

instantiate and loadProperties

At some later point the context will be rehydrated and the projection will need to be rehydrated. First the projection will need to be instantiated, and accordingly, the instantiate method would be called. For our MyNetwork class this may be something like this:

```
protected T instantiate(Context<?> context, Map<String, Object> properties) {
        String name = (String) properties.get(NAME_KEY);
        return new MyNetwork(name);
}
```

This simply fetches the projection's name (which will be stored in ProjectionDryer's loadProperties method - another reason why it's important to call the super's method!) and then creates the network with that name. Next the network's properties need to be loaded, which, according to our addProperties method will be like so:

```
protected void loadProperties(Context<?> context, MyNetwork net, Map<String, Object>
properties) {
        super.loadProperties(context, proj, properties);
        ArrayList<Object[]> edges = (ArrayList<Object[]>) properties.get("edges");
        for (Object[] edgeProperties : edges) {
                net.addEdge(edgeProperties[0], edgeProperties[1], (Double) edgeProperties[2]);
        }
}
```

In this method we first are calling the super's load properties method (so that things like listeners and any other settings will be loaded), then we fetch the edge information from the properties map (the ArrayList... = line), and finally we add the edges into the given network. Note that we may have to cast the properties if the network requires specific types in its methods (for instance if the weight is a Double or double as in this example).

Now the Projection will be added into the context and the rehydration process will continue on.

Registering the new dryer

The different kind of dryers need to be registered in different kinds of ways. If you create a custom FreezeDryer, you must register that dryer to any FreezeDryer instance you create. If you have created a custom ProjectionDryer, you must register it to the ProjectionDryer class by calling its addProjectionDryer method..

File Formats

This page last changed on Nov 23, 2010 by collier.

File Formats

user_path.xml

```
<model name="Model Name">
 <classpath>
   <entry path="..." annotations="[true|false]" />
   <agents path="..." filter="..." />
   <builtin fullname="..." />
 </classpath>
</model>
```

- path is path statement to a directory or jar file. If the directory contains jars then all those jars are added to the classpath. If the directory or its children contain .class files then the directory is added to the classpath. More than one path can be specified. Multiple paths should be "," separated. For .class files, the path functions as a classpath and so it must be the directory immediately above the directories corresponding to the class packages. For example, given the class repast.simphony.MyClass and the directory structure X/bin/repast/simphony/MyClass.class, the path should be X/bin. This applies to path in both entry and agents elements.

- annotations specified whether or not any classes found in the specified path(s) should be processed for annotations. The default is false.

- agents specifies the path where agent classes can be found. These are automatically processed for annotations.

- filter specifies a class filter on the agents path(s). For example a filter of "anl.gov.Agent" will filter out all classes but anl.gov.Agent as agent classes. Filters can contain the wildcard "*". So for example, "anl.gov.*Agent" will include any class in the package anl.gov whose class name ends with Agent.

- the builtin element is for the case when a user needs to add an agent class existing in one of the repast plugins. The user specifies the canonical class name as the fullname.

(Note: For the builtins, since the class can be in any one of the repast plugins, without hard coding the path, it is difficult to use the filter mechanism on all the plugins. This means that unless we can figure out a way to figure out the path of a resource containing a package, we must specify each individual class that we'd like to be considered an agent class. The good news is that this is not a common usage.)

context.xml

context.xml contains the context heirarchy information.

```
<context id="..." class="...">
  <attribute id="..." value="..." type="[int|long|double|float|boolean|string|...]"
display_name="..." converter="..."/>

  <projection id="..." type="[network|grid|continuous space|geography|value layer]">
    <attribute id="..." .../>
  </projection>

  <context id="..." class="...">
     ...
  </context>
</context>
```

Context Element

Attribute Name	Description	Required
id	Unique identifer for the context	YES
class	Fully qualitifed name of a Context implementation. If this is present this context will be used instead of the default	NO

Attribute Element

Attribute Name	Description	Required
id	Unique identifer for the attribute	YES
value	Default value of the attribute	YES
type	The primitive or class type of the attribute	YES
display_name	Optional name used to display the attribute when it is used as a model parameter	NO
converter	Optional implementation of StringConverter used to convert the attribute to and from a string representation	NO

For example,

```
  <context id="MyContext">
    <attribute id="numAgents" value="20" type="int" display_name="Initial Number of Agents"/>
  </context>
```

Projection Element

Attribute Name	Description	Required
id	Unique identifer for the projection	YES
type	The projection type (network, grid, geography, continuous space, value layer)	YES

"Magic" Attributes

If certain "magic" attributes are present, then the context.xml file can be used to instantiate the actual context hierarchy. The attributes are defined in the repast.simphony.dataLoader.engine.AutoBuilderConstants.

Context Attributes

Attribute Id	Description	Allowed values	type	Required
*_timeUnits_*	The tick unit type	Any string that can be parsed by Amount.valueOf	string	NO

Grid Attributes

Attribute Id	Description	Allowed values	type	Required
Any attribute of int type	Any int attributes will be used as the dimensions of the grid. These will be processed in order such that the first becomes the width (x) dimensions, the next the height (y) and so on.	Any int	int	YES
border rule	Specifies the behavior of agents when moving over a grid's borders.	bouncy, sticky, strict, or periodic	string	YES
allows multi	whether or not the grid allows multiple agents in each cell	true or false (default is false)	boolean	NO

For example,

```
<context id="MyContext">

  <projection id="MyGrid" type="grid">
    <attribute id="width" type="int" value="200"/>
    <attribute id="height" type="int" value="100"/>
    <attribute id="border rule" type="string" value="periodic" />
    <attribute id="allows multi" type="boolean" value="true"/>
  </projection>
</context>
```

This will create a 200x100 grid with an id of "MyGrid." The border rule for the grid is "periodic" so the grid will wrap, forming a torus. The grid will also allow multiple agents in each cell.

Continuous Space Attributes

Attribute Id	Description	Allowed values	type	Required
Any attribute of int or double type	Any int or double attributes will be used as the dimensions of the grid. These will be processed in order such that the first becomes the width (x) dimensions, the next the height (y) and so on.	Any int or double	int or double	YES
border rule	Specifies the behavior of agents when moving over a space's borders.	bouncy, sticky, strict, or periodic	string	YES

```
<context id="MyContext">

  <projection id="MySpace" type="continuous space">
    <attribute id="width" type="int" value="200"/>
    <attribute id="height" type="int" value="100"/>
    <attribute id="border rule" type="string" value="strict" />
  </projection>
</context>
```

This will create a 200x100 continuous space with an id of "MySpce." The border rule for the grid is "strict" so
any movement across the border will cause an error.

Network Space Attributes

Attribute Id	Description	Allowed values	type	Required
directed	Whether or not the network is directed	true or false. Default is false	boolean	NO
edge class	The fully qualified name of a class that extends RepastEdge. Any edges created by the network will be of this type.	Any properly formatted class name extending RepastEdge	string	NO

```
<context id="MyContext">

  <projection id="MyNetwork" type="network">
    <attribute id="directed" type="boolean" value="true"/>
  </projection>
</context>
```

This will create a network with an id of "MyNetwork." The network will be directed and use the default RepastEdge as the edge type.

GUI Parameters and Probes

This page last changed on Nov 19, 2007 by etatara.

GUI Parameters

GUI Parameters are the simulation parameters that appear in the parameters view in the application GUI. These parameters are automatically created from the attribute elements in a score file or from any @Parameters method annotations that are used in a Context. At the moment, these two options only allow for the creation of unconstrained numeric, boolean and String parameters. However, constrained parameters of more complicated types can be defined using an xml based format. The first section of this document describes these extended parameters, the second describes how to save, load and "set as default" parameters using the parameters tools menu, and the last section looks at the parameters returned from agent probes.

Extended Parameters

Extended parameters are defined in an external xml file that must be named "extended_params.xml". If such a file is found in the scenario directory then the parameters defined therein are created and displayed in the parameters panel together with the parameters created from the score file. The format for this file is as follows:

```
<parameters>
  <parameter name="*name*" displayName="*display name*" type="*type*" defaultValue="*default*"
values="*values list*"/>
    ...
</parameters>
```

The file consists of a "<parameters>" root element which contains one or more parameter definitions. The attributes of this "<parameter>" element are described below:

- *name* - the unique name of the parameter. This is used to retrieve and set parameter values.
- *display name* - the name used to display the parameter in the GUI.
- *type* - the fully qualified java type of the parameter. Acceptable short-hands are int, double, float, boolean, and string for the primitives and String types.
- *default* - the default value of the parameter.
- *values list* - an **optional** list of parameter values. If this list is present then the legal parameter values will be constrained to this list and the GUI then displays these values in a combo-box rather than a text field. The list must be space delimited and any string values that contain a space must be enclosed in single quotes (e.g. 'my parameter'). **Note** that if any of the string values are single quoted then they all must be.

Some example parameters:

```
<parameters>
  <parameter name="names" displayName="Family Names" type="string" defaultValue="Cormac"
values="Cormac Nicola Caitrin"/>
  <parameter name="numbers" displayName="A Constrained List of Numbers" type="int"
defaultValue="2" values="2 10 3 5"/>
</parameters>
```

Convertors and custom parameter types

Repast Simphony can internally handle the conversion of numeric and boolean types to and from the Strings in the gui display and the parameter file. However, if you wish to use a type other than these, then you must provide a converter that will convert an Object to and from a String representation. A converter is class that implements StringConverter and is used to create the appropriate Object from a string representation when reading the xml file and taking input from the gui. The converter also converts the object to a string representation. This string is then used when writing the object out and displaying the parameter's value in the gui.

For example, supposing you wish use the following Name class as a Parameter.

```
package repast.test.viz;public class Name {

  private String first, last;
  private int hashCode;

  public Name(String first, String last) {
    this.first = first;
    this.last = last;
    hashCode = 17;
    hashCode = 37 * hashCode + first.hashCode();
    hashCode = 37 * hashCode + last.hashCode();
  }

  public String getFirst() {
    return first;
  }

  public String getLast() {
    return last;
  }

  public int hashCode() {
    return hashCode;
  }

  public boolean equals(Object obj) {
    if (obj instanceof Name) {
      Name other = (Name) obj;
      return other.first.equals(this.first) && other.last.equals(this.last);
    }
    return false;
  }

  public String toString() {
    return "Name(" + first + ", " + last + ")";
  }
}
```

Your parameter xml then would look something like:

```
<parameters>  ...
  <parameter name="name" displayName="Full Name" type="repast.test.viz.Name" defaultValue="Nick
Collier"
             converter="repast.test.viz.NameConverter" />
</parameters>
```

The type is the fully qualified name of the parameter type (in this case "Name") and the converter is the fully qualified name of the converter class. The converter must implement the StringConverter interface. StringConverter has two methods for converting to and from a String representation.

```
package repast.parameter;

/**
 * Converts an Object to and from a String representation. Subclasses
 * are specialized for specified objects.
 *
 * @author Nick Collier
 */
public interface StringConverter<T> {

  /**
   * Converts the specified object to a String representation and
   * returns that representation. The representation should be such
   * that <code>fromString</code> can recreate the Object.
   *
   * @param obj the Object to convert.
   *
   * @return a String representation of the Object.
   */
  String toString(T obj);

  /**
   * Creates an Object from a String representation.
   *
   * @param strRep the string representation
   * @return the created Object.
   */
  T fromString(String strRep);
}
```

The NameConverter then looks like:

```
package repast.test.viz;

import repast.parameter.*;

public class NameConverter implements StringConverter<Name> {

  /**
   * Converts the specified object to a String representation and
   * returns that representation. The representation should be such
   * that <code>fromString</code> can recreate the Object.
   *
   * @param obj the Object to convert.
   * @return a String representation of the Object.
   */
  public String toString(Name obj) {
    return obj.getFirst() + " " + obj.getLast();
  }

  /**
   * Creates an Object from a String representation.
   *
   * @param strRep the string representation
   * @return the created Object.
   */
  public Name fromString(String strRep) {
    int index = strRep.indexOf(" ");
    String first = strRep.substring(0, index);
    String last = strRep.substring(index + 1, strRep.length());
    return new Name(first, last);
  }
}
```

It is also possible to use the values attribute to constrain the possible values of a converted parameter. For example,

```
<parameters>  ...
```

```
  <parameter name="name" displayName="Full Name" type="repast.test.viz.Name" defaultValue="Nick
Collier"
             converter="repast.test.viz.NameConverter" values="'Nick Collier' 'David Beckham'
'Wayne Rooney'"/>

</parameters>
```

When constraining a converted parameter, the elements in the values list must be Strings that can be converted using the converter. If you do constrain a converted parameter in this way, you must override equals (and hashCode) in your custom type. If not, then the constraint validation will most likely fail.

Parameters Panel Tools

The menu on the parameters panels allows the user to save and load parameter values, and set the current parameter values as the default.

1. Saving - clicking the save menu item will prompt for a file name and then save the current parameter values into that file.
2. Loading - clicking the load menu item will prompt for a file name and then load the parameter values from that file into the current parameters. **This assumes that the current parameters are compatible with those being loaded, that is, that the names and types of parameters in the file match the current parameters.**
3. Set as Default - this allows the user to set current parameter values as defaults such that these values will be used when the simulation is reset. Clicking this menu item prompts for which parameters to set as defaults and then sets those as the new defaults.

Agent Probes

An agent can be probed by clicking on its visual representation in the gui. The agent's parameters will then be displayed in a probe panel. By default, the displayed parameters are derived from an agent's Java bean properties which are themselves derived from get / set accessor method pairs. For example,

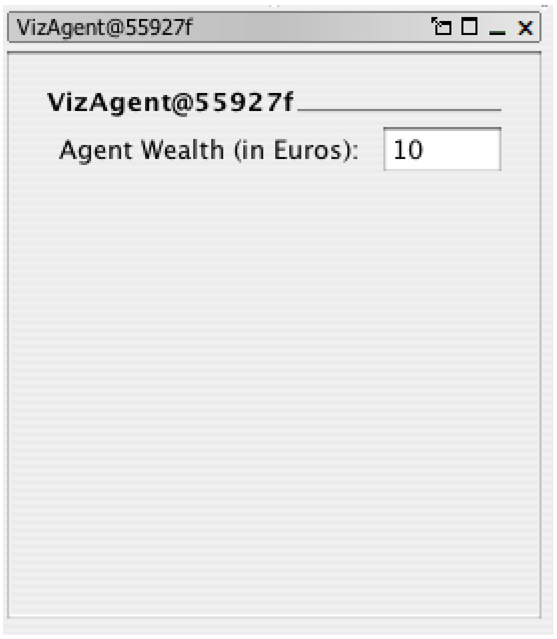
```
public class VizAgent {
  ...
  public double getWealth() {
    ...
  }

  public void setWealth(double val) {
    ...
  }
}
```

This creates a "wealth" parameter. The probe panel will display an entry for "wealth" whose value is the result of the "getWealth" call. When the user enters a new value in the "wealth" text field then the setWealth method will passed this new value. Read-only parameters can be created by specifying only a get-style method. The label for this field will be the simple java bean properties name.

Which properties are treated as parameters, and how to display them can be further refined by adding an @Parameter annotation to either the getter or setter of a property method pair. (The information in the annotation applies equally to both methods.) Using a Parameter annotation, a more descriptive display name can be specified. For example,

```
public class VizAgent {
  ...
  @Parameter(usageName="wealth", displayName="Agent Wealth (in Euros)")
  public double getWealth() {
    ...
  }

  public void setWealth(double val) {
    ...
  }
}
```

The usageName should correspond to the java bean property which is typically the method name minus the get/set with the first character lower-cased. The displayName specifies the gui label for this parameter in the probe window.



Note that if any method is annotated with @Parameter, then only @Parameter annotated methods will be displayed in the probe panel.

Using the @Parameter annotation, you can also specify the name of a converter to convert to and from a displayable String. By specifying a converter, any type of object contained by the agent can be displayed as a parameter. This converter is identical that described above in the extended parameters section. See the convertors and custom parameter types section above for more details. For example,

```
public class VizAgent {
  ...
  @Parameter(usageName="wealth", displayName="Agent Wealth (in Euros)",
converter="repast.demo.MoneyConverter")
  public Money getWealth() {
    ...
  }

  public void setWealth(Money val) {
    ...
  }
}
```

In this case, the agent's wealth is represented by a Money class and a MoneyConverter is used to convert to and from a String value.

Lastly, by default the probe's title will be the result of calling "toString()" on the probed object (e.g. VizAgent@55927f, in the above screen shot). A more appropriate title can be specified either by overriding toString() and returning the desired value, or annotating any method that returns the desired title with @ProbeID. For example,

```
public class VizAgent {
  ... @ProbeID public String name()  {  return name;
}
  @Parameter(usageName="wealth", displayName="Agent Wealth (in Euros)",
converter="repast.demo.MoneyConverter")
  public Money getWealth() {
    ...
  }

  public void setWealth(Money val) {
    ...
  }
}
```

Random Numbers

This page last changed on Nov 19, 2007 by etatara.

In Repast 3, the uchicago.src.sim.util.Random class was the primary point for working with random numbers and the random number stream. In Simphony you now have the capability of having multiple random streams with separate seeds through the RandomRegistry and the RandomHelper class.

RandomRegistry

Like many of the other registries in Simphony, the RandomRegistry is available through the current RunState (through its getRandomRegistry() method). This interface provides the capability for storing named random distributions and retrieving them. It also can set and retrieve the seed that a stream will use (or is using) for streams it creates through its addStream(String, Class, ...) methods.

RandomHelper

The RandomRegistry by itself is somewhat difficult to work with, to simplify using it the RandomHelper class was created. This provides methods for creating the random streams that will be used, and for retrieving the streams after they have been created, when you do not have a reference to the current RunState.

Through its getStream, getUniform and getNormal methods, you can retrieve distributions from the current RandomRegistry. Through its numerous createXXX methods, you can create the distributions and store them to the current RandomRegistry.

The RandomHelper also handles creating and maintaining a default Uniform number stream. This stream is the one used by Repast when it needs a random number, and is no different then the other Uniform number streams, except that it will always exist when requested from the RandomHelper (when the simulation is running). If you wish to use a Uniform distribution for your random numbers, you can avoid some of the extra work that comes with the named streams (meaning, specifying the name of the stream, and possibly casting the stream) by simply using RandomHelpers' getDefault, nextInt, and nextDouble methods. You also can retrieve and/or set the seed for the default stream through the RandomHelper's getSeed and setSeed methods.

Usage

Generally there are going to be two types of usage of the random functionality. The first would be when you just need random numbers of some sort. In this case you would just want to use RandomHelper's default methods, meaning the getDefault(), nextInt and nextDouble methods.

The other case is when you want numbers from a non-Uniform distribution, or you want multiple random streams functioning at once. In this case you can use RandomHelper's default methods for any uniform numbers, but for producing multiple streams or a non-Uniform stream you have to use the methods that take in a stream name. Each of these separate cases will be walked through next.

Single Default Stream Usage

When working with the default Uniform stream there is really only one thing you may want to look at manipulating, the default stream's random seed. This is set through the RandomHelper's setSeed(int) method. Setting the random seed should generally happen when the model is setting up. There are two ways you can do this. One way is in a ControllerAction, but if you wish to use the random numbers in setting up the model you will not have your seed set ahead of time. The other (correct) way is in a ParameterSetter.

In the ParameterSetter you are going to want to call RandomHelper's setSeed method, as below:

```
public class MySimpleSetter extends SimpleParameterSetter {
        public void next(Parameters params) {
                RandomHelper.setSeed(System.currentTimeMillis());
        }
}
```

The above code will set the default uniform distribution's seed to be the current time. Now that the time is set, the agents or controller actions (in their runInitialize methods) can use the default distribution and will recieve numbers based off of that seed. For example if you had an agent that wanted to move based on some random condition, that could look like:

```
public void agentAction() {
        if (myHappiness < RandomHelper.nextDouble()) {
                move();
        }
}
```

The above example compares the agent's happiness to a random number between 0 and 1, in (0, 1), if the random number is greater than the agent's happiness it will move.

Multiple Stream / Non Uniform Stream Usage

As with the single streams, you generally are going to want to create your Random number streams when your model is initializing, primarily in a ParameterSetter, at the point the ParameterSetter will be executed the run's RandomRegistry will be setup, and you can feel free to use the RandomHelper.

With these streams you are going to need to consistently use the same stream name(s) when working with the RandomHelper or the RandomRegistry. You will need to pass this name into the methods, so that that the helper/registry knows where to get the values from. For example, let's say that you want 2 random number streams; one a Poisson distribution, and one a Normal distribution. With both of these using a distinct random seed.

So an example initialization would look like:

```
public class MySetter extends ParameterSetter {
        public void next(Parameters params) {
                // first create the normal distribution
         // we pass in null so that it will use the current random registry
         RandomHelper.createNormal(null, "MyNormal", normalSeed, 0, 1);
                // next we create the poisson distribution
```

```
                  RandomHelper.createPoisson(null, "MyPoisson", poissonSeed, 0);
                }
        }
```

The above code created a normal distribution named "MyNormal" that has a mean of 0, a standard deviation of 1, and some seed (normalSeed). It then creates a poisson distribution named "MyPoisson" with a mean of 0 and some specified seed (poissonSeed). It is worth noticing however that unlike the default stream, if you do not setup these streams with the given name they will **not** be created for you. You must create them in some way (for instance like above) or you will not be able to use them.

You can then use these distributions from your code. If you wanted to do the same thing as the single stream example, but with the normal stream you would do:

```
public void agentAction() {
        if (myHappiness < RandomHelper.getNormal("MyNormal").nextDouble()) {
                move();
        }
}
```

The Poisson stream is slightly less straighforward to use, in that direct helper methods are not provided that will cast the stream to a Poisson distribution. For this distribution you must cast the stream yourself. This is more complicated than it sounds, and functions simply like:

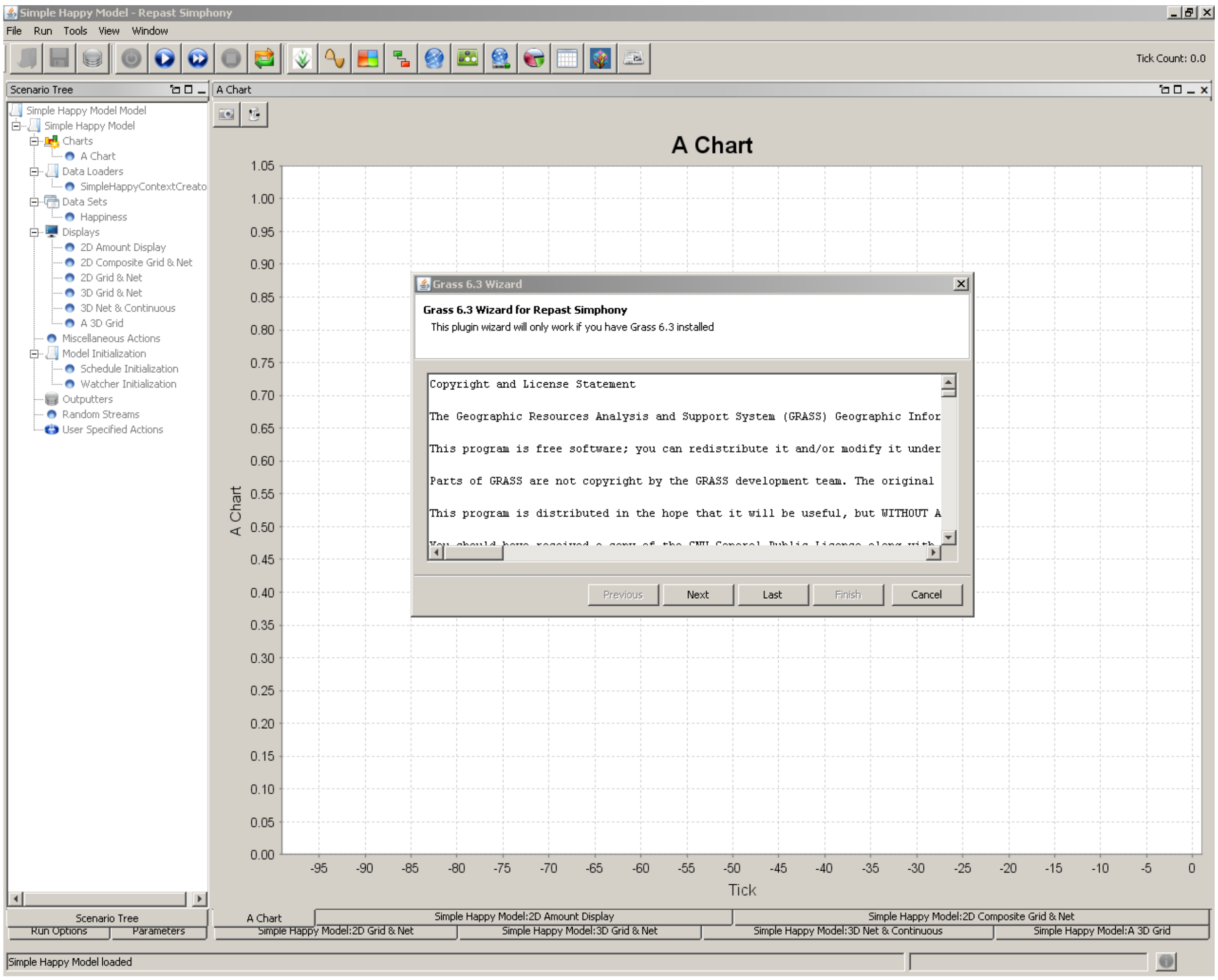
```
public void agentAction() {
        // grab the distribution with the given name, casting it to a Poisson distribution
 Poisson randomStream = (Poisson) RandomHelper.getStream("MyPoisson");
        // now use it
 if (myHappiness < randomStream.nextDouble()) {
                move();
        }
}
```

Running Grass within a Reapst Simphony Project

This page last changed on Dec 11, 2008 by altaweel.

Running Grass 6.3 within a Reapst Simphony Project Launching Grass Script in Repast Simphony In Repast Simphony, Grass scripts can be called throughout a Repast Simphony execution or post-modeling analysis, similar to R, Matlab, and other analysis plugins, and can evolve or apply your Grass-configured data via script commands. The Grass plugin can be found at repast.simphony.grass. For launching a Grass application via the Repast Simphony GUI, follow the steps listed below.

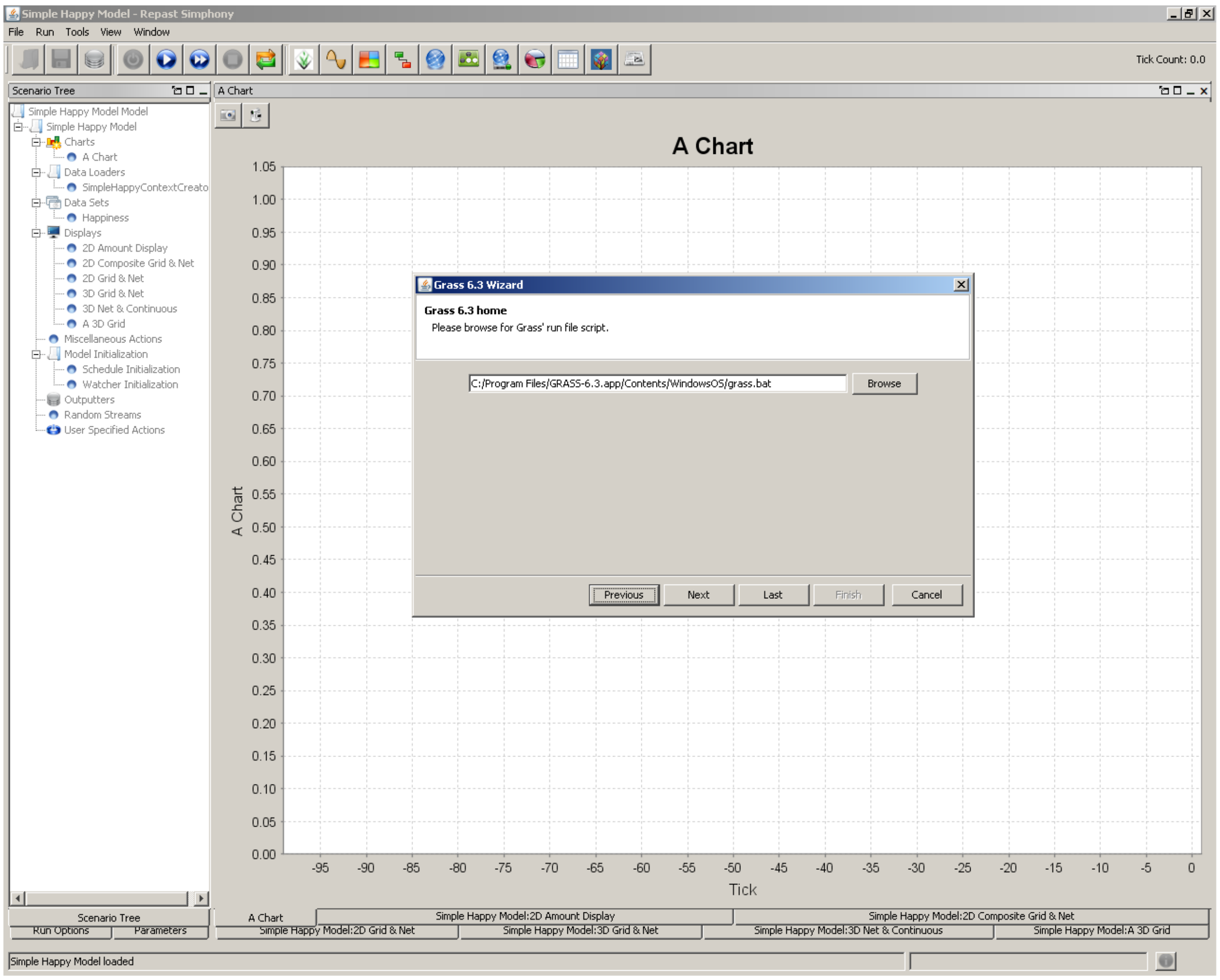
1. Press the Grass button ( ) or choose the Repast Simphony Tool option ("Run Grass") for Grass.



Grass 6.3 Wizard for Repast Simphony
This plugin wizard will only work if you have Grass 6.3 installed

```
Copyright and License Statement

The Geographic Resources Analysis and Support System (GRASS) Geographic Infor

This program is free software; you can redistribute it and/or modify it under

Parts of GRASS are not copyright by the GRASS development team. The original

This program is distributed in the hope that it will be useful, but WITHOUT A
```

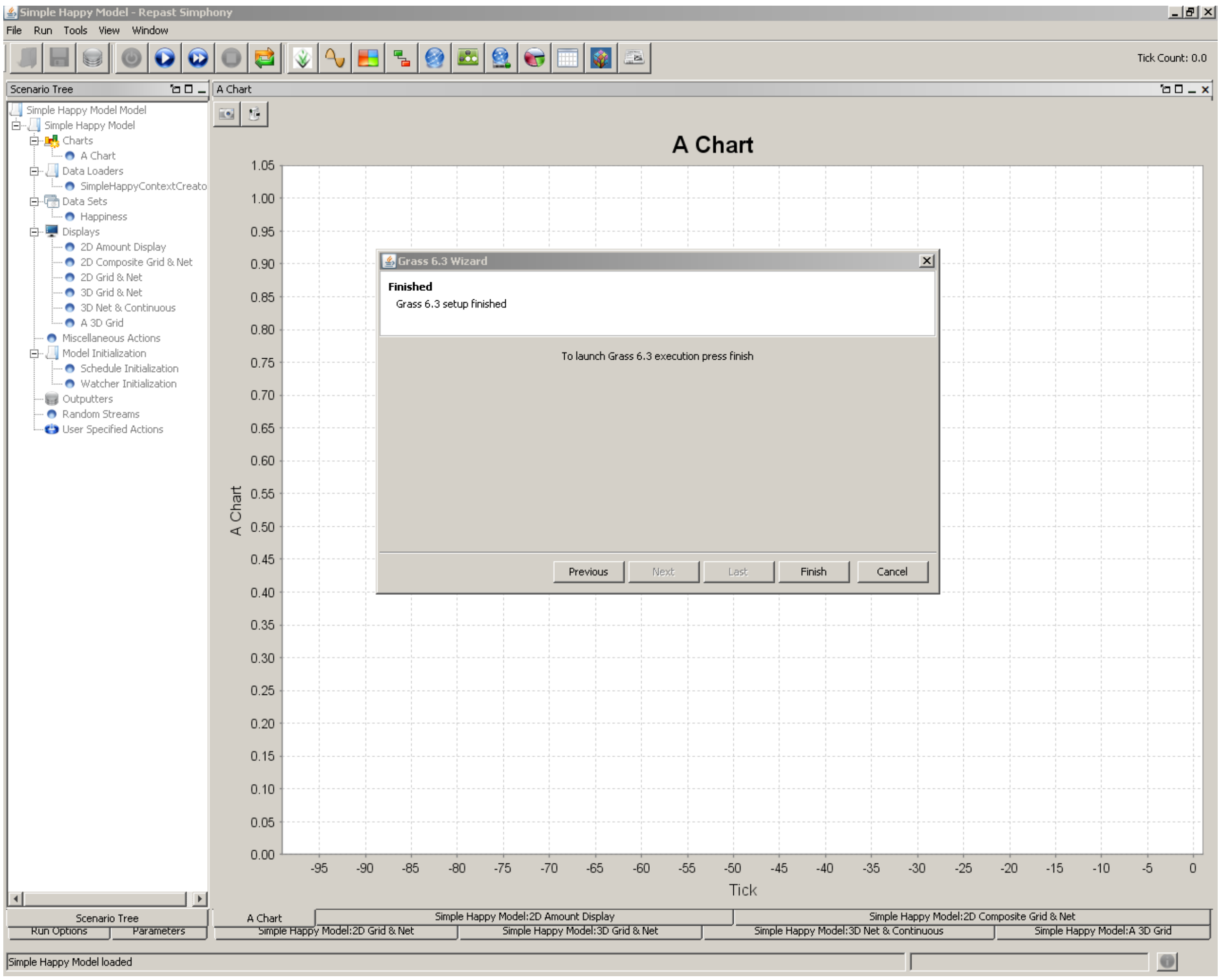
2. Go to the next step, which should allow you to choose the Grass run script you desire via the script chooser.



Grass 6.3 home
Please browse for Grass' run file script.

C:/Program Files/GRASS-6.3.app/Contents/WindowsOS/grass.bat

3. Go to the final step. Press the Finish button. This should launch your script.



Finished
Grass 6.3 setup finished

To launch Grass 6.3 execution press finish

Simple Happy Model loaded

Alternatively, you can launch your work via a script command within your Repast Simphony project. Simply create a LaunchScript object, which takes a String argument that represents your launch command, then call launch(). For an example of this, look at the /test folder in the repast.simphony.grass plugin. Two example shell scripts are also included in the /testScripts folder in the repast.simphony.grass plugin. These scripts need to be configured for your machine in order to be used. For further information on Grass, please go to the Grass site (http://grass.osgeo.org/).

Running GridGain on a Repast Simphony Project

This page last changed on Nov 22, 2010 by north.

Using repast.simphony.grid

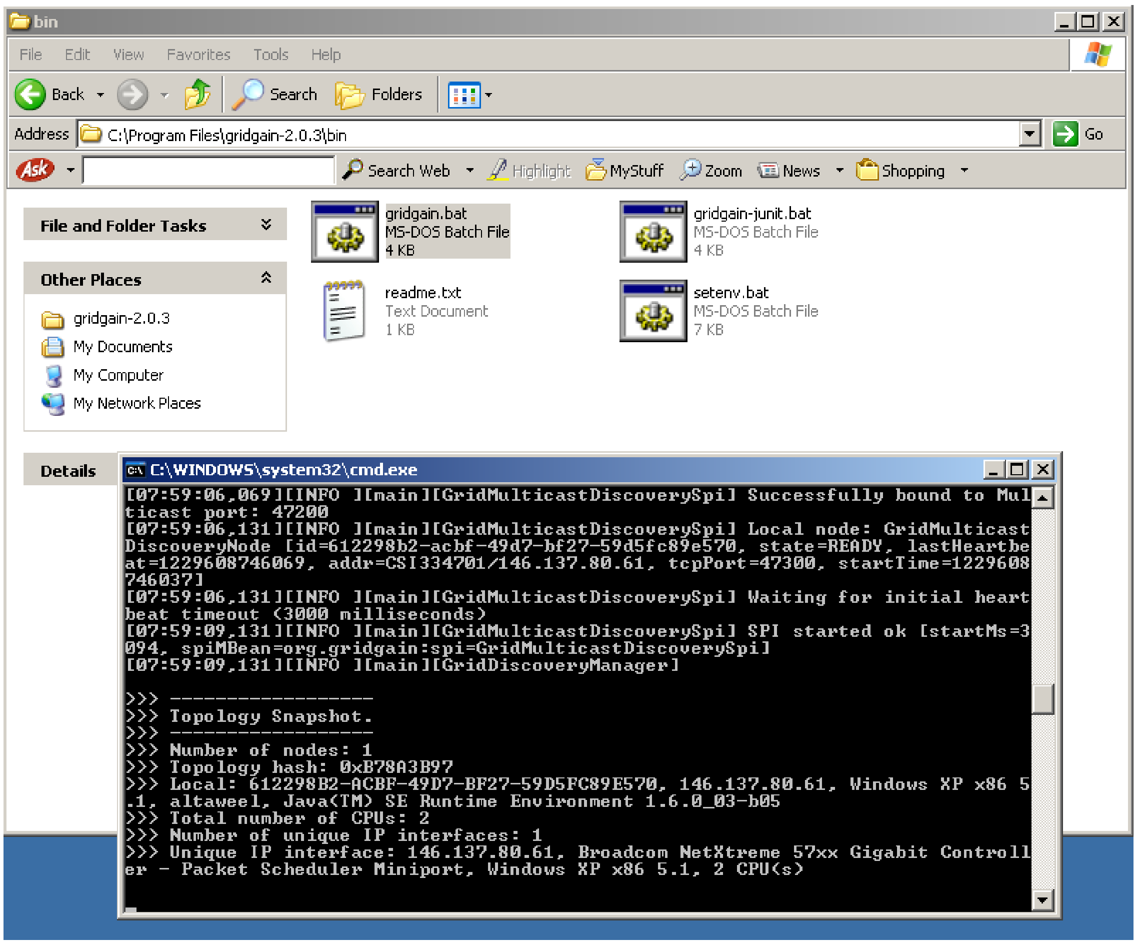
The plugin repast.simphony.grid does not come with the standard Repast S installation; however, it can be obtained via svn download. This plugin uses GridGain (http://www.gridgain.com/), an open source tool allowing for grid computing. Using this plugin in Repast S, you may need to reconfigure your project so that it can be distributed. Detailed instructions and examples of GridGain can be found at the site link provided.

Setting Up a GridGain Repast S Project

The plugin repast.simphony.grid comes with a demo folder that provides an example project that can be distributed on multiple machines. The project simply has agents moving to random locations on a grid and stating their id. No matter how many nodes the project is executed on, the agents will state their ids and proceed to move within the grid. To enable GridGain, you will first need to download the project and install it on all the machines you intend to use it on, including the machine that you are working on.

The demo folder of repast.simphony.grid will be used as an example project to explain how a Repast S project can use GridGain. First, launch the GridGain script file, which should be a file called gridgain.bat on Windows or gridgain.sh on Linux/Unix machines. This file is located in the bin folder of the GridGain installation. Simply launch this file on the remote nodes you plan to use. To use many nodes, you can simply automate this process through a custom script file that launches all the nodes you would like to use.

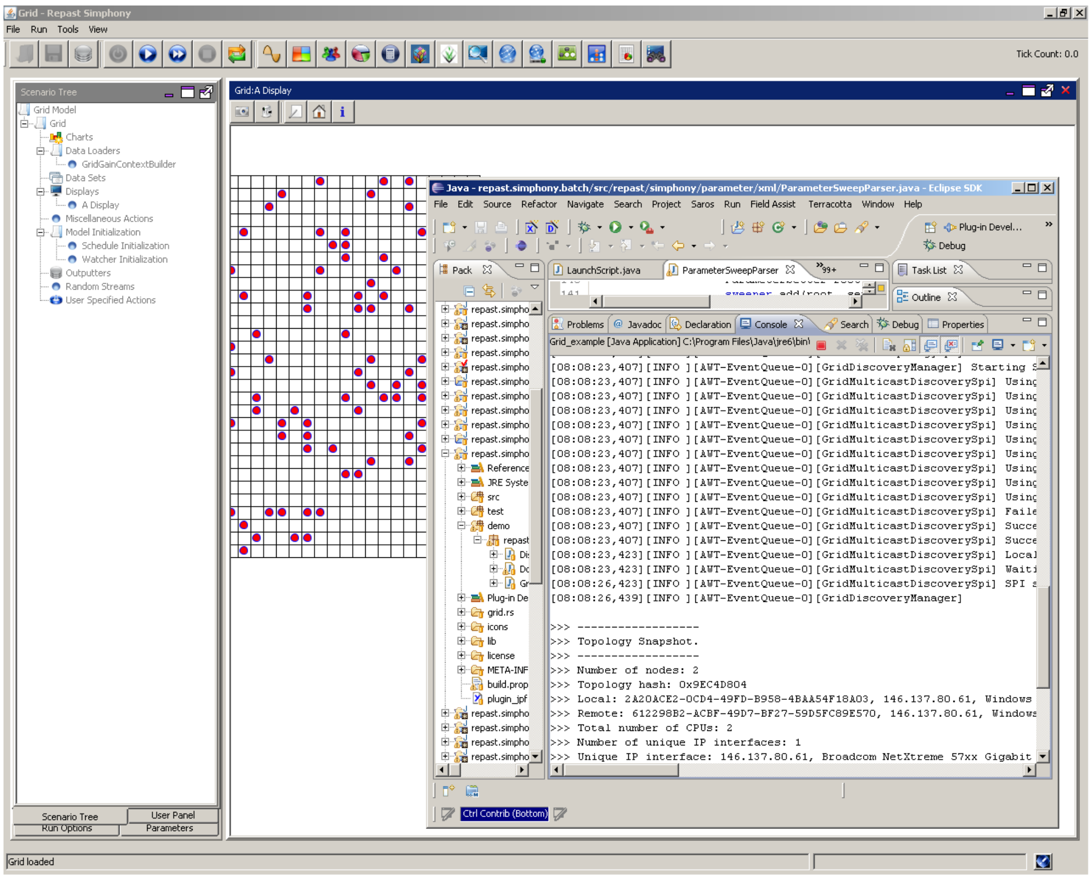
1. Launching a GridGain script on a node.



```
[07:59:06,069][INFO ][main][GridMulticastDiscoverySpi] Successfully bound to Mul
ticast port: 47200
[07:59:06,131][INFO ][main][GridMulticastDiscoverySpi] Local node: GridMulticast
DiscoveryNode [id=612298b2-acbf-49d7-bf27-59d5fc89e570, state=READY, lastHeartbe
at=1229608746069, addr=CSI334701/146.137.80.61, tcpPort=47300, startTime=1229608
746037]
[07:59:06,131][INFO ][main][GridMulticastDiscoverySpi] Waiting for initial heart
beat timeout (3000 milliseconds)
[07:59:09,131][INFO ][main][GridMulticastDiscoverySpi] SPI started ok [startMs=3
094, spiMBean=org.gridgain:spi=GridMulticastDiscoverySpi]
[07:59:09,131][INFO ][main][GridDiscoveryManager]

>>> ------------------
>>> Topology Snapshot.
>>> ------------------
>>> Number of nodes: 1
>>> Topology hash: 0xB78A3B97
>>> Local: 612298B2-ACBF-49D7-BF27-59D5FC89E570, 146.137.80.61, Windows XP x86 5
.1, altaweel, Java(TM) SE Runtime Environment 1.6.0_03-b05
>>> Total number of CPUs: 2
>>> Number of unique IP interfaces: 1
>>> Unique IP interface: 146.137.80.61, Broadcom NetXtreme 57xx Gigabit Controll
er - Packet Scheduler Miniport, Windows XP x86 5.1, 2 CPU(s)
```

After your nodes are launched, then simply configure a normal execution of a Repast S project. If you are using the Eclipse IDE, for example, configure the run exactly how you would for all Repast S runs; however, be sure to have all the jars from the repast.simphony.grid plugin in your classpath. After this, you are ready to launch a distributed project.

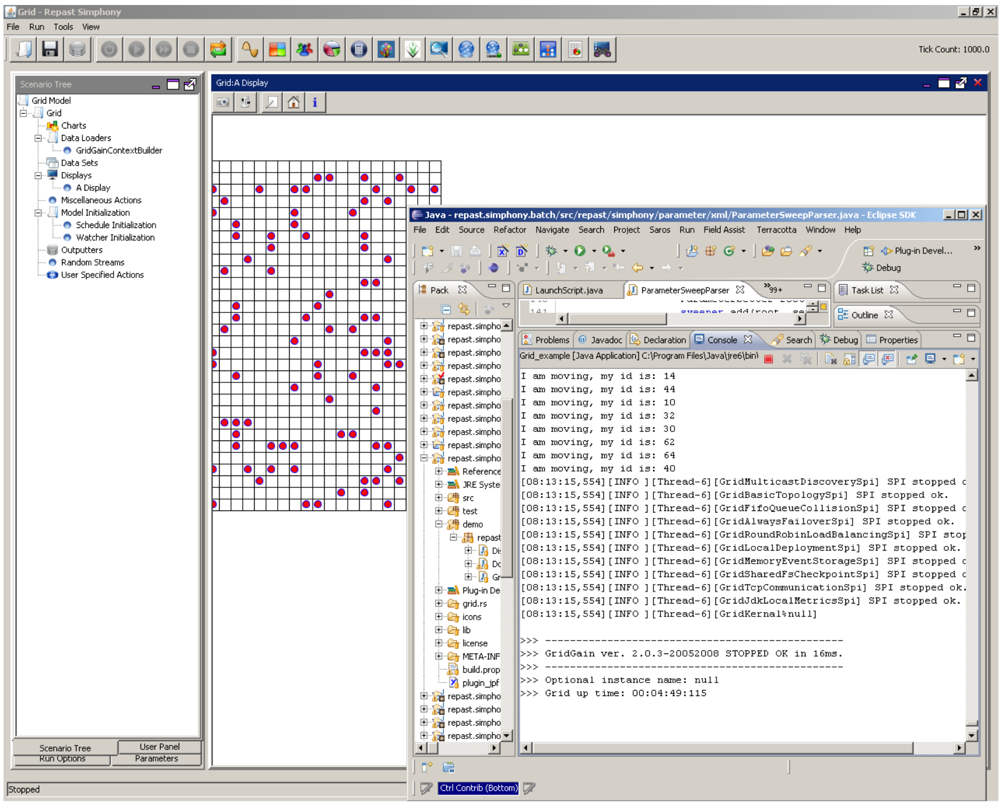
2. The demo project from repast.simphony.grid being launched. The image shows the Repast S GUI and the console from Eclipse. Notice that the console states two nodes are being used in this example, although both are on the same machine. Distributed processes on multiple machines would work exactly as the same as the example shown.



```
Grid_example [Java Application] C:\Program Files\Java\jre6\bin\
[08:08:23,407][INFO ][AWT-EventQueue-0][GridDiscoveryManager] Starting S
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Using
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Faile
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Succe
[08:08:23,407][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Succe
[08:08:23,423][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Local
[08:08:23,423][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] Waiti
[08:08:26,423][INFO ][AWT-EventQueue-0][GridMulticastDiscoverySpi] SPI s
[08:08:26,439][INFO ][AWT-EventQueue-0][GridDiscoveryManager]

>>> ------------------
>>> Topology Snapshot.
>>> ------------------
>>> Number of nodes: 2
>>> Topology hash: 0x9EC4D804
>>> Local: 2A20ACE2-0CD4-49FD-B958-4BAA54F18A03, 146.137.80.61, Windows
>>> Remote: 612298B2-ACBF-49D7-BF27-59D5FC89E570, 146.137.80.61, Windows
>>> Total number of CPUs: 2
>>> Number of unique IP interfaces: 1
>>> Unique IP interface: 146.137.80.61, Broadcom NetXtreme 57xx Gigabit
```

At this point, simply run the project and that will enable the work to be distributed. The example project should run for 1000 ticks. At the end of the run, you should get a message that the distributed process has stopped.

3. Image showing the console and the Repast GUI at the end of the simulation.

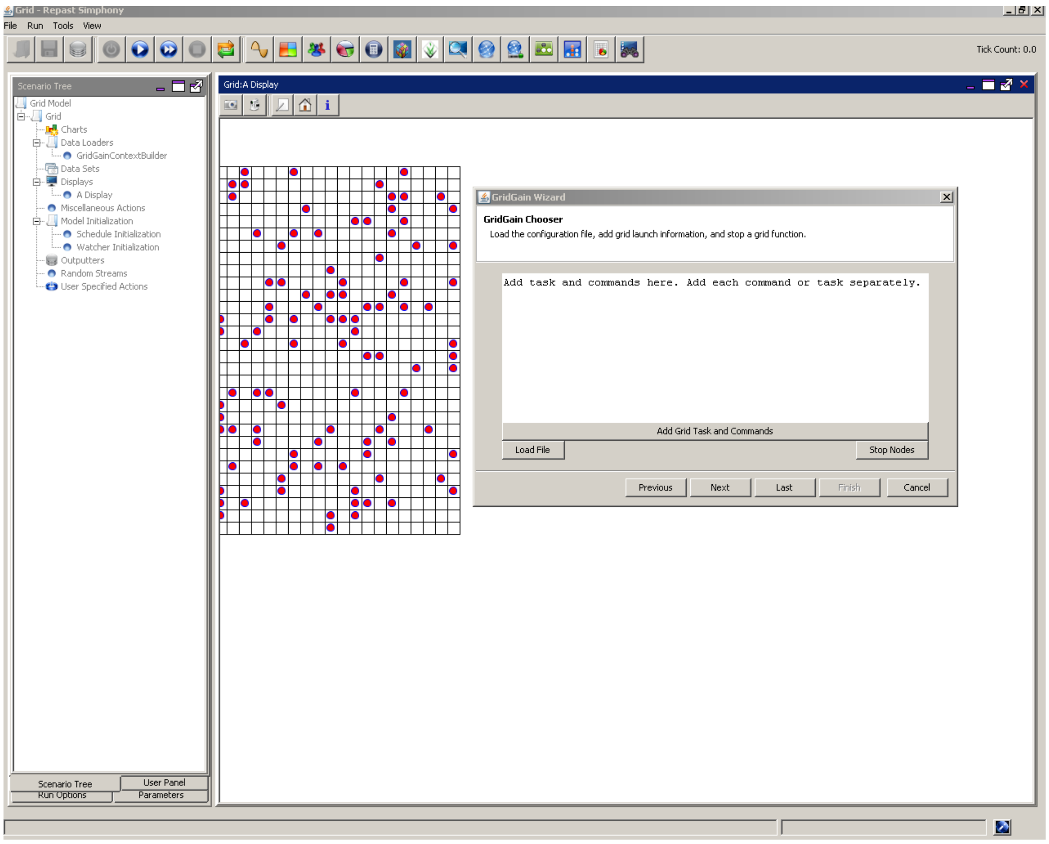


```
Grid_example [Java Application] C:\Program Files\Java\jre6\bin\
I am moving, my id is: 14
I am moving, my id is: 44
I am moving, my id is: 10
I am moving, my id is: 32
I am moving, my id is: 30
I am moving, my id is: 62
I am moving, my id is: 64
I am moving, my id is: 40
[08:13:15,554][INFO ][Thread-6][GridMulticastDiscoverySpi] SPI stopped c
[08:13:15,554][INFO ][Thread-6][GridBasicTopologySpi] SPI stopped ok.
[08:13:15,554][INFO ][Thread-6][GridFifoQueueCollisionSpi] SPI stopped c
[08:13:15,554][INFO ][Thread-6][GridAlwaysFailoverSpi] SPI stopped ok.
[08:13:15,554][INFO ][Thread-6][GridRoundRobinLoadBalancingSpi] SPI stop
[08:13:15,554][INFO ][Thread-6][GridLocalDeploymentSpi] SPI stopped ok.
[08:13:15,554][INFO ][Thread-6][GridMemoryEventStorageSpi] SPI stopped c
[08:13:15,554][INFO ][Thread-6][GridSharedFsCheckpointSpi] SPI stopped c
[08:13:15,554][INFO ][Thread-6][GridTcpCommunicationSpi] SPI stopped ok.
[08:13:15,554][INFO ][Thread-6][GridJdkLocalMetricsSpi] SPI stopped ok.
[08:13:15,554][INFO ][Thread-6][GridKernal%null]

>>> ---------------------------------------------
>>> GridGain ver. 2.0.3-20052008 STOPPED OK in 16ms.
>>> ---------------------------------------------
>>> Optional instance name: null
>>> Grid up time: 00:04:49:115
```

GridGain Wizard

In addition to running this project via the standard run in Eclipse, you can also launch project using the plugin wizard for GridGain. Press the ( ) button, which will provide standard licensing information in the initial window when started. The next window will allow you to enter arguments and configuration file, if needed, for executing your processes. For example, you may enter the name of the task (e.g., GridGain.task) and additional commands (i.e., name of object used for jobs being distributed). GridGain also uses files to help configure remote nodes, which can be loaded here as well. Look at the GridGain site for further instructions on configuring a network using a file.

4. Wizard window showing place to add tasks, commands, and/or network configuration files.



GridGain Chooser

Load the configuration file, add grid launch information, and stop a grid function.

```
Add task and commands here. Add each command or task separately.
```

When all the arguments have been entered into the wizard window, then simply press the "Finish" button at the end to start the execution of the commands given.

Additional Information

GridGain can be configured and setup in a variety of different ways. How you configure your project and divide work tasks will have a lot to do with how fast your project will run. Be sure to investigate different network configurations and task divisions in order to increase the efficiency of your project. This may involve extending the GridGainTask object in repast.simphony.grid.setup. Further information on GridGain can be found at the website for the tool.

Running Terracotta on a Repast Simphony Project

This page last changed on Dec 11, 2008 by altaweel.

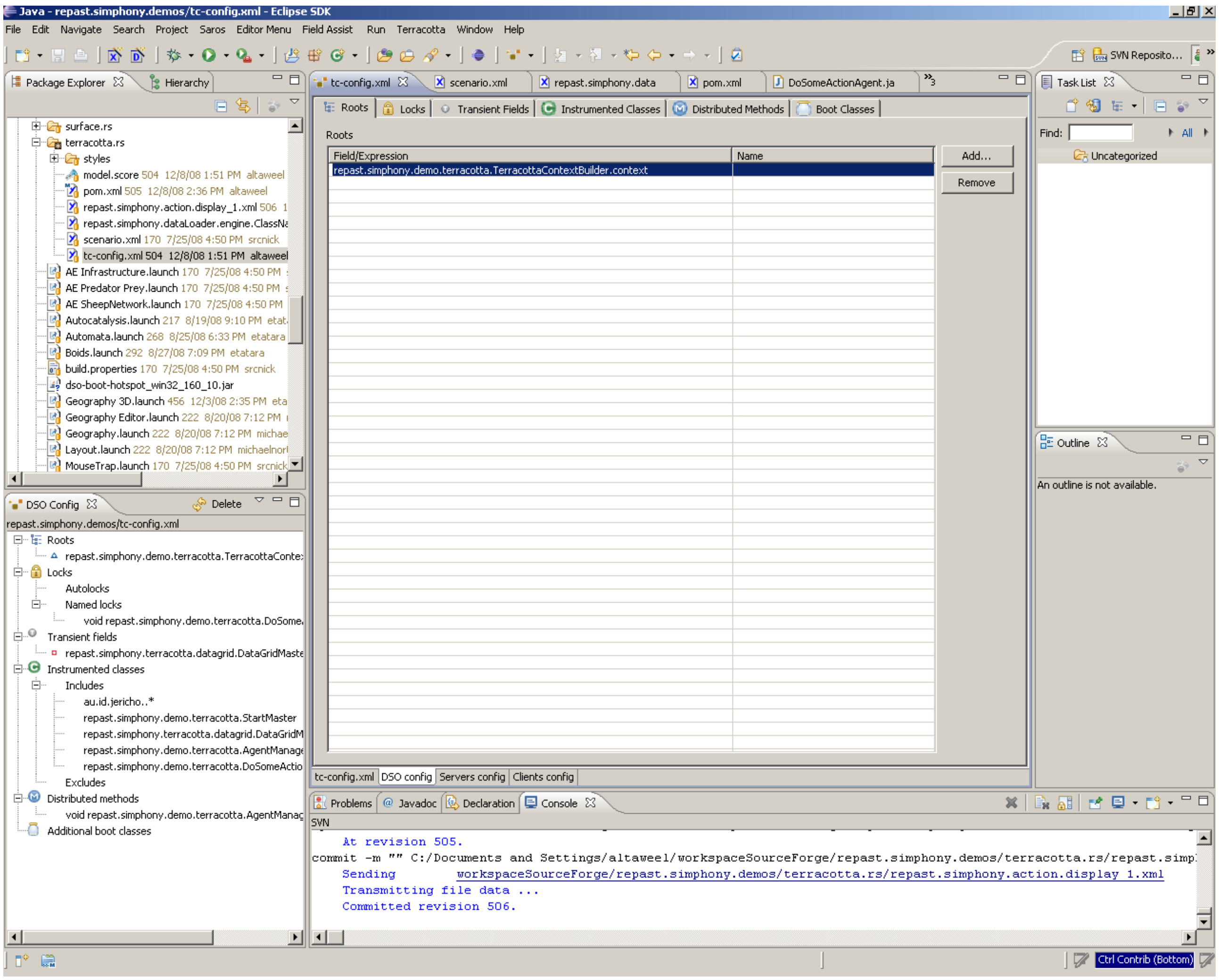
Introduction to Terracotta in Repast Simphony

Repast Simphony includes a recently developed plugin (repast.simphony.terracotta) that is intended to be used for distributing projects on a network cluster. The plugin assumes that the user has downloaded the open source Terracotta (http://www.terracotta.org/) tool and installed it on the user's machine and nodes that will be used

Setup in Eclipse

Most users will need to configure their project in an IDE in order to be distributed. Terracotta has a plugin that can be coupled into Eclipse (http://www.terracotta.org/confluence/display/orgsite/Download#Download-EclipsePlugin). Once the plugin is installed, you can enable any Eclipse project to run using Terracotta. Follow the instructions for the Terracotta setup in Eclipse (http://www.terracotta.org/confluence/display/docs1/Eclipse+Plugin+Reference+Guide) in order to distribute your project. The following steps need to be implemented in order to allow the user to run Terracotta within Eclipse:

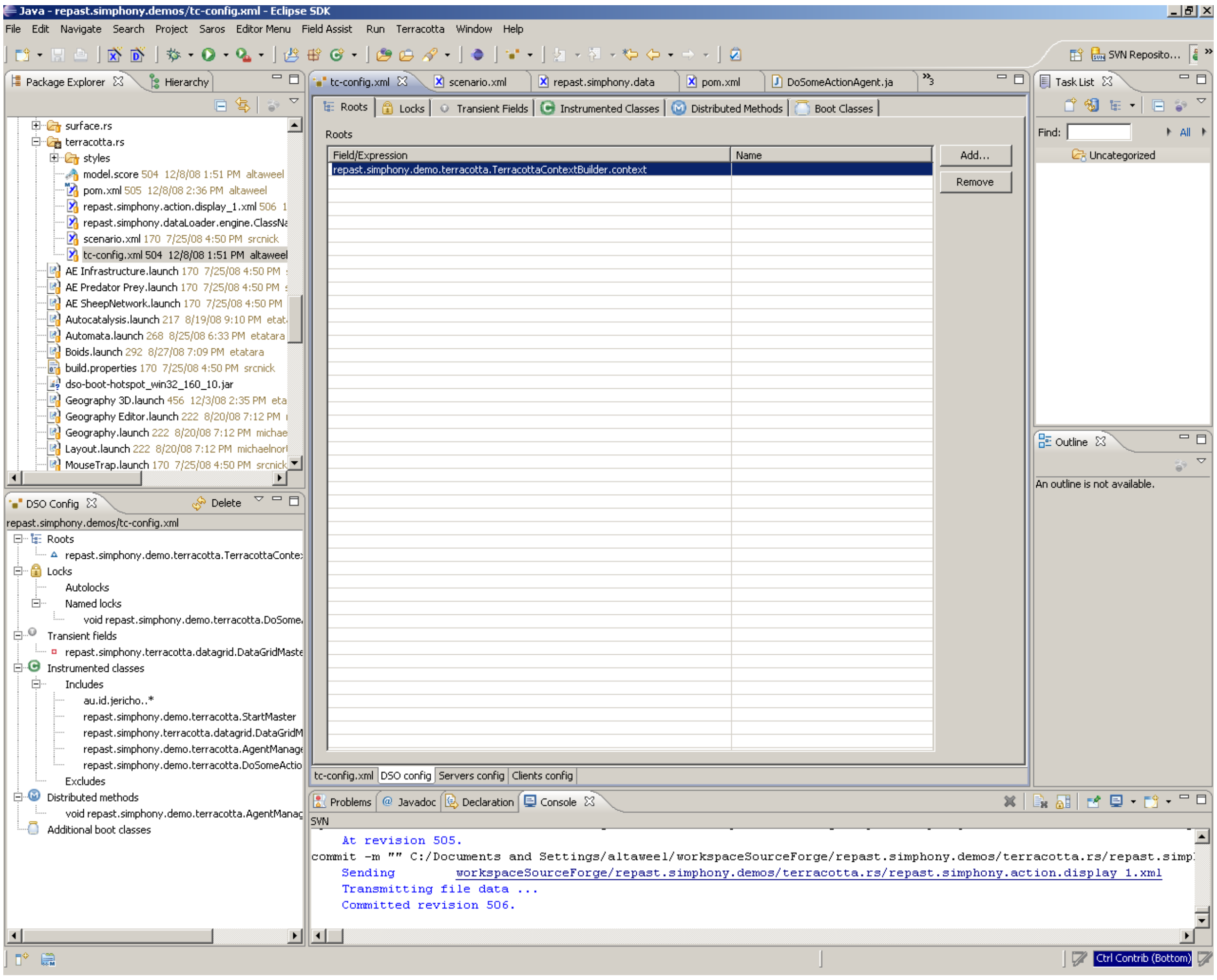
1. Enable terracotta nature on your project. Choose the Terracotta menu item in Eclipse and create the configuration file.



Field/Expression	Name
repast.simphony.demo.terracotta.TerracottaContextBuilder.context	

```
SVN
    At revision 505.
commit -m "" C:/Documents and Settings/altaweel/workspaceSourceForge/repast.simphony.demos/terracotta.rs/repast.simp
    Sending        workspaceSourceForge/repast.simphony.demos/terracotta.rs/repast.simphony.action.display_1.xml
    Transmitting file data ...
    Committed revision 506.
```

2. Setup your Terracotta configuration file in order to distribute methods and data; the file is generally entitled tc-config.xml. See http://www.Terracotta.org/confluence/display/docs1/Eclipse+Plugin+Reference+Guidefor further information on basic setup for distributing a standard Java project.



Java - repast.simphony.demos/tc-config.xml - Eclipse SDK

```
terracotta.rs
  styles
  model.score 504  12/8/08 1:51 PM  altaweel
  pom.xml 505  12/8/08 2:36 PM  altaweel
  repast.simphony.action.display_1.xml 506
  repast.simphony.dataLoader.engine.ClassNa
  scenario.xml 170  7/25/08 4:50 PM  srcnick
  tc-config.xml 504  12/8/08 1:51 PM  altaweel
AE Infrastructure.launch 170  7/25/08 4:50 PM
AE Predator Prey.launch 170  7/25/08 4:50 PM
AE SheepNetwork.launch 170  7/25/08 4:50 PM
Autocatalysis.launch 217  8/19/08 9:10 PM  etat
Automata.launch 268  8/25/08 6:33 PM  etatara
Boids.launch 292  8/27/08 7:09 PM  etatara
build.properties 170  7/25/08 4:50 PM  srcnick
dso-boot-hotspot_win32_160_10.jar
Geography 3D.launch 456  12/3/08 2:35 PM  eta
Geography Editor.launch 222  8/20/08 7:12 PM
Geography.launch 222  8/20/08 7:12 PM  michae
Layout.launch 222  8/20/08 7:12 PM  michaelnor
MouseTrap.launch 170  7/25/08 4:50 PM  srcnick
```

DSO Config

```
repast.simphony.demos/tc-config.xml
Roots
  repast.simphony.demo.terracotta.TerracottaConte
Locks
  Autolocks
  Named locks
    void repast.simphony.demo.terracotta.DoSome
Transient fields
  repast.simphony.terracotta.datagrid.DataGridMaste
Instrumented classes
  Includes
    au.id.jericho..*
    repast.simphony.demo.terracotta.StartMaster
    repast.simphony.terracotta.datagrid.DataGridM
    repast.simphony.demo.terracotta.AgentManage
    repast.simphony.demo.terracotta.DoSomeActio
  Excludes
Distributed methods
  void repast.simphony.demo.terracotta.AgentManag
Additional boot classes
```

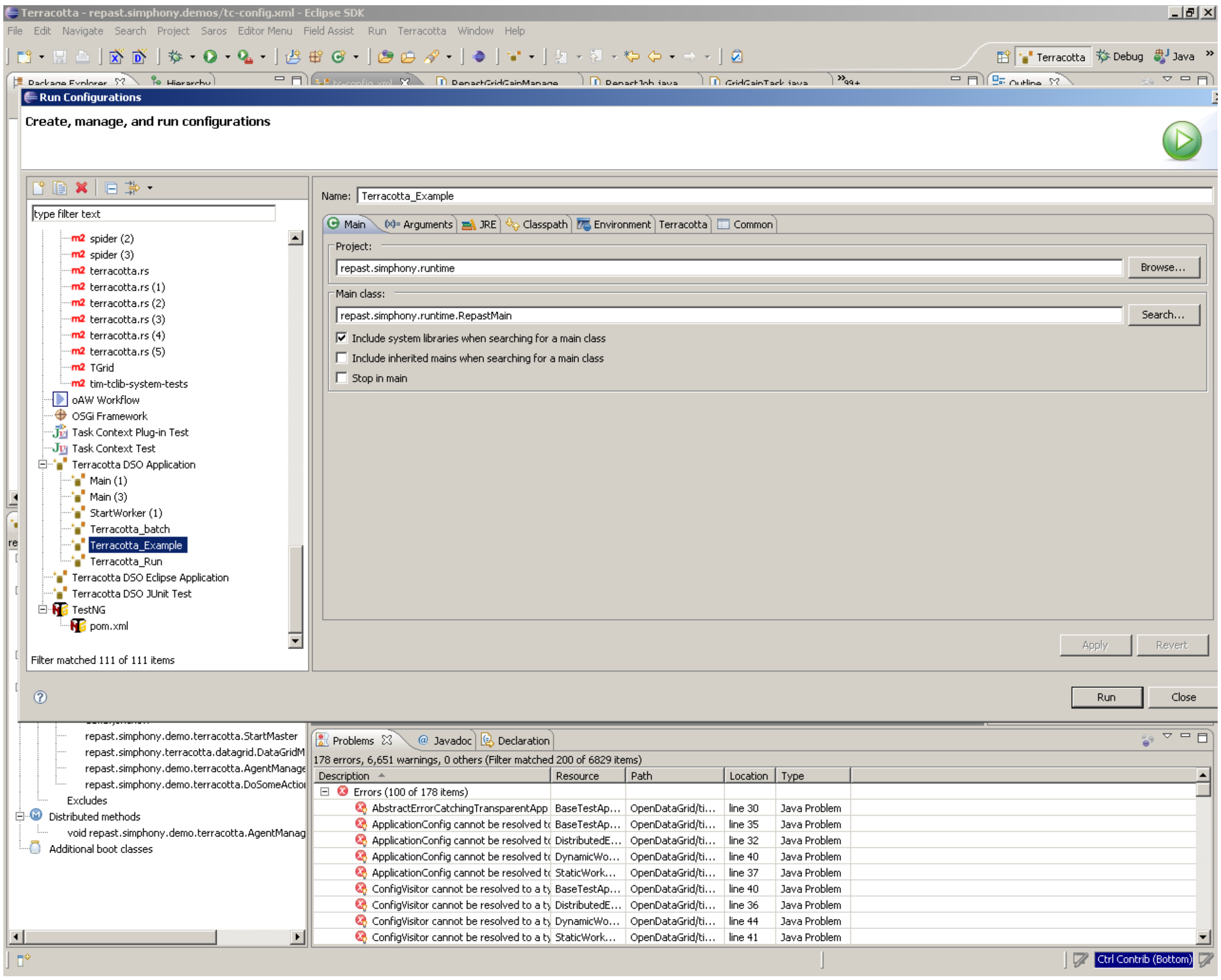
Roots

Field/Expression	Name
repast.simphony.demo.terracotta.TerracottaContextBuilder.context	

Console — SVN

```
    At revision 505.
commit -m "" C:/Documents and Settings/altaweel/workspaceSourceForge/repast.simphony.demos/terracotta.rs/repast.simp
    Sending        workspaceSourceForge/repast.simphony.demos/terracotta.rs/repast.simphony.action.display 1.xml
    Transmitting file data ...
    Committed revision 506.
```

3. Setup the Terracotta DSO application run in order to run the project and investigate results. This can be setup similar to a normal java application run, except you need to make sure you are pointing to the Terracotta tc-config.xml file. Terracotta should appear as a choice option in the run execution window.


Terracotta - repast.simphony.demos/tc-config.xml - Eclipse SDK

Run Configurations

Create, manage, and run configurations

type filter text

```
m2 spider (2)
m2 spider (3)
m2 terracotta.rs
m2 terracotta.rs (1)
m2 terracotta.rs (2)
m2 terracotta.rs (3)
m2 terracotta.rs (4)
m2 terracotta.rs (5)
m2 TGrid
m2 tim-tclib-system-tests
oAW Workflow
OSGi Framework
Task Context Plug-in Test
Task Context Test
Terracotta DSO Application
    Main (1)
    Main (3)
    StartWorker (1)
    Terracotta_batch
    Terracotta_Example
    Terracotta_Run
Terracotta DSO Eclipse Application
Terracotta DSO JUnit Test
TestNG
    pom.xml
```

Filter matched 111 of 111 items

Name: Terracotta_Example

Main | Arguments | JRE | Classpath | Environment | Terracotta | Common

Project: repast.simphony.runtime

Main class: repast.simphony.runtime.RepastMain

[✓] Include system libraries when searching for a main class
[ ] Include inherited mains when searching for a main class
[ ] Stop in main

```
repast.simphony.demo.terracotta.StartMaster
repast.simphony.terracotta.datagrid.DataGridM
repast.simphony.demo.terracotta.AgentManage
repast.simphony.demo.terracotta.DoSomeActio
Excludes
Distributed methods
    void repast.simphony.demo.terracotta.AgentManag
Additional boot classes
```

Problems | Javadoc | Declaration

178 errors, 6,651 warnings, 0 others (Filter matched 200 of 6829 items)

Description	Resource	Path	Location	Type
Errors (100 of 178 items)				
AbstractErrorCatchingTransparentApp	BaseTestAp...	OpenDataGrid/ti...	line 30	Java Problem
ApplicationConfig cannot be resolved t	BaseTestAp...	OpenDataGrid/ti...	line 35	Java Problem
ApplicationConfig cannot be resolved t	DistributedE...	OpenDataGrid/ti...	line 32	Java Problem
ApplicationConfig cannot be resolved t	DynamicWo...	OpenDataGrid/ti...	line 40	Java Problem
ApplicationConfig cannot be resolved t	StaticWork...	OpenDataGrid/ti...	line 37	Java Problem
ConfigVisitor cannot be resolved to a ty	BaseTestAp...	OpenDataGrid/ti...	line 40	Java Problem
ConfigVisitor cannot be resolved to a ty	DistributedE...	OpenDataGrid/ti...	line 36	Java Problem
ConfigVisitor cannot be resolved to a ty	DynamicWo...	OpenDataGrid/ti...	line 44	Java Problem
ConfigVisitor cannot be resolved to a ty	StaticWork...	OpenDataGrid/ti...	line 41	Java Problem

Utilizing a Master/Worker Pattern in Repast Simphony

In many simulations, the basic Terracotta setup may not be sufficient to significantly boost simulation execution speed. It is also possible that you may not need or desire a Master/Worker configuration. In such cases, simply creating the tc-config.xml file might be sufficient to launching your project. However, a process similar to a Master/Worker network configuration might be necessary in order to maximize speed, assign processes to different nodes, and execute methods efficiently. The repast.simphony.terracotta plugin utilizes the Terracotta Master/Worker pattern, which also incorporates standard Terracotta. Users can look at the simple example in repast.simphony.demo.terracotta to configure their own project. Below is an explanation on applying the Master/Worker pattern in a Repast Simphony project. Please look at Terracotta's Master/Worker design pattern (http://www.terracotta.org/web/display/orgsite/Master+Worker) for further information.

In the package repast.simphony.terracotta.datagrid, the DataGridMaster class enables the Master to be applied using routing ids, though you will likely only use one Master id, and a list of work items (i.e. processes for distribution). The first class of importance is RepastWork, which is an interface that uses CommonJ's Work class. This interface should be used for functions or agents that need to be distributed within the cluster. That is, this interface informs Master/Worker that the object is a work item used in distribution. Users should also start the worker nodes using Terracotta's DynamicWorkerFactory, DefaultWorker, or other similar classes. The cluster nodes are "workers" that execute submitted processes (i.e. instances implementing the RepastWork interface). The example in repast.simphony.demo.terracotta shows a simple project that applies repast.simphony.terracotta along with Terracotta's Master/Worker objects.

Configuring and Running

You can apply projects on remote machines using Maven and/or a standard Terracotta launch configuration. In order to use Maven within Terracotta in remote nodes, you should first setup Maven properly in your Terracotta folder on the remote machines. Go to your Terracottaa Maven directory (e.g.:../.m2/repository/maven-tc-plugin/). The Terracotta Repast Simphony project you create can be installed on other machines using Maven. If you use Maven, install the Maven plugin for Terracotta first by using the "mvn install" command. Then, go into the directory containing your Terracotta Repast Simphony application. Invoke "mvn package" to build the application properly. After you have installed your project, including the tc-config.xml file within your project, on other nodes, you are now ready to execute your project either using Repast Simphony's GUI interface or batch configuration. If you are using Maven, start the Terracotta server via the "mvn tc:start" command. To start the Master, invoke "mvn -DactiveNodes=master tc:run" command. Then start the Workers via "mvn -DactiveNodes=worker tc:run." For using DSO in Terracotta, start the Master and Worker nodes using Terracotta DSO, which can be done within Eclipse. Below are examples of how to launch these:

Linux/Unix: sh dso-java.sh -cp path-to-bin-directory -Dtc.config=nameOfTerracottaConfigFile pathToClassWithMainMethod

Windows: dso-java.bat -cp path-to-bin-directory -Dtc.config=nameOfTerracottaConfigFile pathToClassWithMainMethod

Launch your project using your Terracotta DSO Application run configuration from the machine you are working. At this point, your project should be running on the cluster you have setup. If you are launching Terracotta from a script file, in the Repast Simphony GUI mode you can launch this script file by simply clicking on the Terracotta button ( ) or pull down menu option "Run Terracotta" in the Tools menu. For further information on configuring Terracotta for your system(s), please see Terracotta's Deployment Guide (http://www.terracotta.org/web/display/orgsite/Deployment+Guide).

Upgrading Repast Simphony 1.0, 1.1, and 1.2 Models to Version 2.0

This page last changed on Nov 22, 2010 by north.

To upgrade your Repast Simphony 1.0, 1.1 or 1.2 model to run in Repast Simphony 2.0, simply do the following:

1. Right click on the scenario folder (usually "yourmodelname.rs").
2. Choose "Convert Scenario" from the popup menu.

This will create a user_path.xml and a context.xml from your old Score file. It will also make backup copy of the scenario.xml file and then remove any displays from the scenario file. Removing the displays is necessary because old Piccolo 2D displays will no longer load in version 2.0.

Using System Dynamics

This page last changed on Nov 22, 2010 by north.

Repast Simphony (Repast S) provides a System Dynamics (SD) implementation as described in the repast.simphony.systemsdynamics.Formula class. To use the Repast S SD tools in the visual editor simply complete the following steps:

1.) Add numeric (e.g., double precision) properties to the agent to reflect the SD variables of interest.

2.) Add a property of type "System Dynamics Equation" to the agent. The automatically provided template is as follows:

new Formula(this, "y", "x * dx", Type.EQUATION)

3.) Replace the "y" output variable with the name of the output property in the agent.

4.) Replace the formula "x * dx" with the formula of interest, using the agent properties, mathamtical functions, and constants. To use the properties of other agents, create an agent property, assign it with the agent of interest, and refer to it in the formula as "otherAgent.otherAgentPropertyName". Please note, that "dx" refers to the elapsed time since the equation was last evaulated. By default the equation will be evaluated once every tick.

5.) Optionally replace Type.EQUATION with Type.EULER as appropriate.

To use the Repast S SD tools in the Java or Groovy simply follow these steps:

1.) Add numeric (e.g., double precision) fields to the agent class to reflect the SD variables of interest.
2.) Add a field of type "Formula" to the agent class.

3.) Assign the Formula as follows:

new Formula(this, "y", "x * dx", Type.EQUATION)

4.) Replace the "y" output variable with the name of the output field in the agent class.

5.) Replace the formula "x * dx" with the formula of interest, using the agent fields, mathamtical functions, and constants. To use the fields of other agent classes, create an agent field, assign it with the agent of interest, and refer to it in the formula as "otherAgent.otherAgentFieldName". Please note, that "dx" refers to the elapsed time since the equation was last evaulated. By default the equation will be evaluated once every tick.

6.) Optionally replace Type.EQUATION with Type.EULER as appropriate.

Using the DataSet Wizard and Editor

This page last changed on Nov 19, 2007 by etatara.

Data sets are used to specify what data you would like collected from your simulation and the metadata tags that are associated with the data (for more on the logging framework in general see the Logging Framework page).

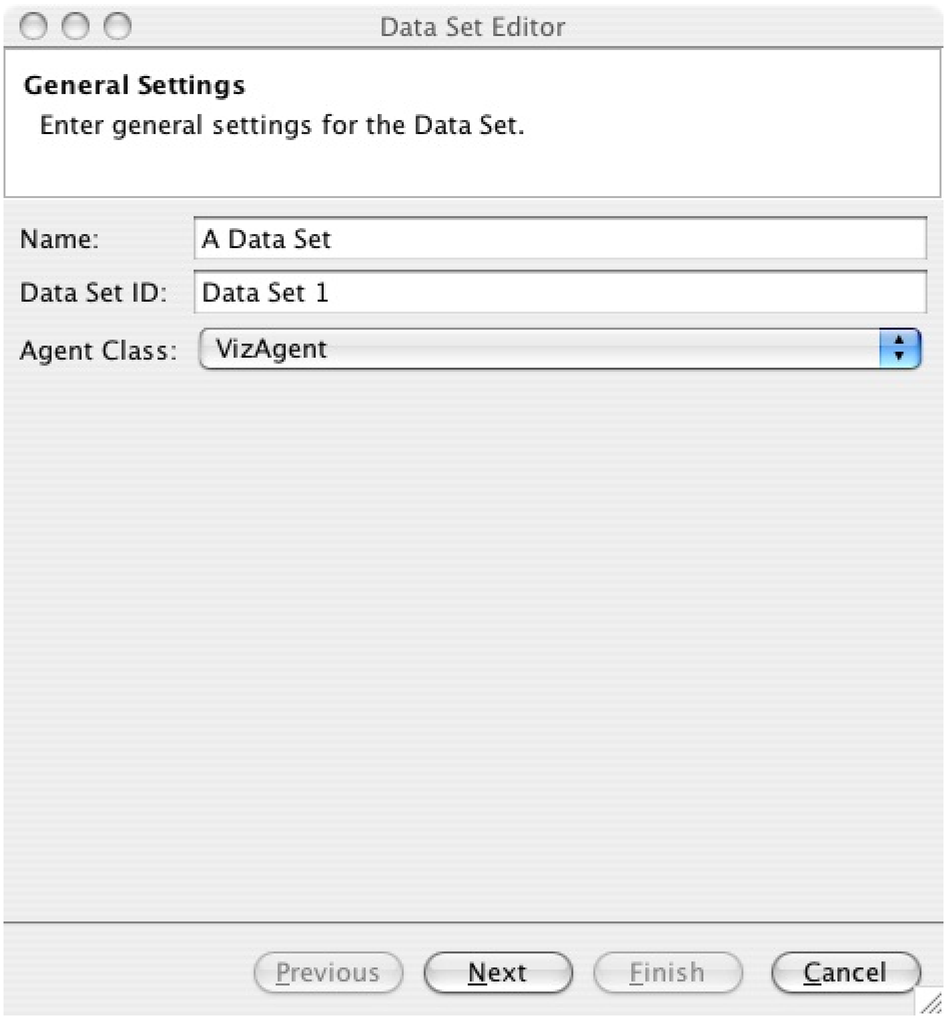
Data Mappings

Simphony's data collection architecture uses a tabular data paradigm where data is collected under specific column headings as in a tabular comma delimited file. The user then is responsible for mapping a data source to a column name. For example, the source that produces the current tick count is typically mapped to the "tick" column.

Using the Wizard

The wizard takes you through the steps to building a data set. After running through the wizard you may re-edit the data set by double clicking on it in the runtime's tree. This will bring up the same set of editing steps as the wizard uses.

Step 1



General Settings

Enter general settings for the Data Set.

Name: A Data Set

Data Set ID: Data Set 1

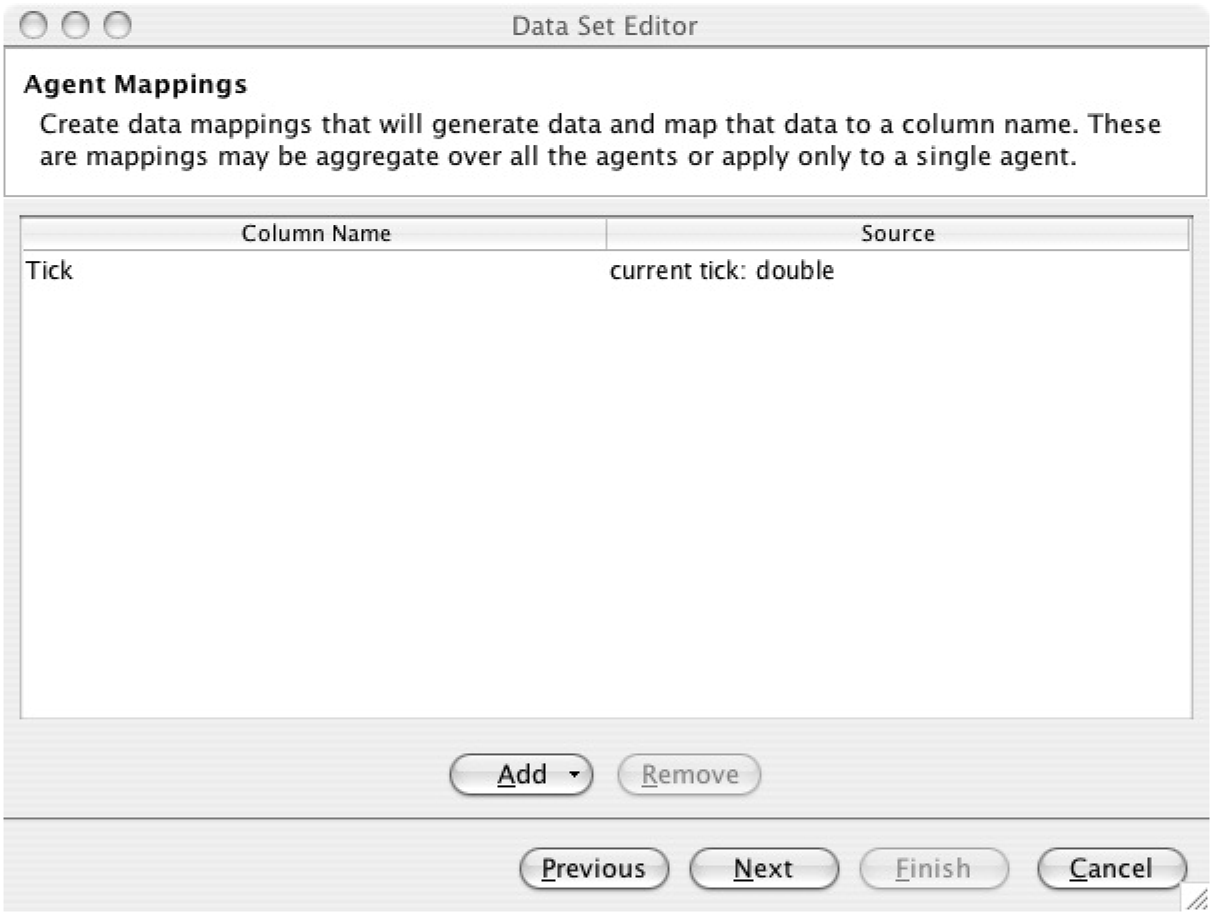
Agent Class: VizAgent

The first step in the wizard lets you specify some information about the data set itself, specifically:

- A Name - this will be used when displaying it in the runtime's tree
- A Data Set Id - this is the id that uniquely identifies this data set from other data sets
- An Agent Class - this specifies the class of agent (or other object) found in the context(s) that data will be collected from

Hit Next to go to the next step in the wizard.

Step 2



Agent Mappings

Create data mappings that will generate data and map that data to a column name. These are mappings may be aggregate over all the agents or apply only to a single agent.

Column Name	Source
Tick	current tick: double

In the second step you specify the data you would like collected. The source of the data can be the tick count, the run number, a method call made on the agents themselves, some aggregate function over a method call on the agents themselves, or a script evaluation. For more on the latter, see formula scripting. Clicking the add button will add a new entry to the data mappings. By default a simple non-aggregate mapping is created. Additional aggregate etc. mapping types can be added via the menu that appears when clicking the "menu triangle" on the add button.

Non-aggregate vs. aggregate mappings

Non-aggregate mappings will be executed on each agent of the class specified in the first step, resulting in as many "rows" of data as there are agents of that type. For example, For example, if you want to collect each agent's name, and the value of the agent's Happiness property, you would create two entries in the interface. If we assume that the agent class has methods that return the agent's name and happiness called getName() and getHappiness() (respectively), to create these two example entries you would:

1. Click the Add button - a new entry will be created in the editor
2. Click on the "Source" cell in the new entry and a list of possible choices will appear
3. Select the one for getName()
4. Click on the "Column Name" cell to name the column for this mapping
5. Repeat steps 1 through 4 but this time Selecting getHappiness()
6. Click on Add to get the default tick entry

You now will have three entries in the table, one for the agent's name, one for the agent's happiness and one for the current tick. Assuming this data set is used to record this data to a file and there are two agents named "John" and "Sam", the result would be something like:

```
 "Tick", "Name", "Happiness"

1.0, "John", 3.0
1.0, "Sam", 3.0
2.0, "John", 1.5
2.0, "Sam", 1.5
```

The point being that the data is recorded for each agent.

In constrast, aggregate mappings perform some aggregating type function over data collected from all the agents. For example, if we want to get the mean of all the agents happiness, we would do:

1. Click the add button to add the default "Tick" entry
2. Click the add button's menu triangle and select "Add Aggregate Mapping"
3. Choose "getHappiness" for the method and "Mean" for the function
4. Click OK
5. Click on the "Column Name" cell (the one that reads "New Column") and replace "New Column" with "Avg. Happiness"

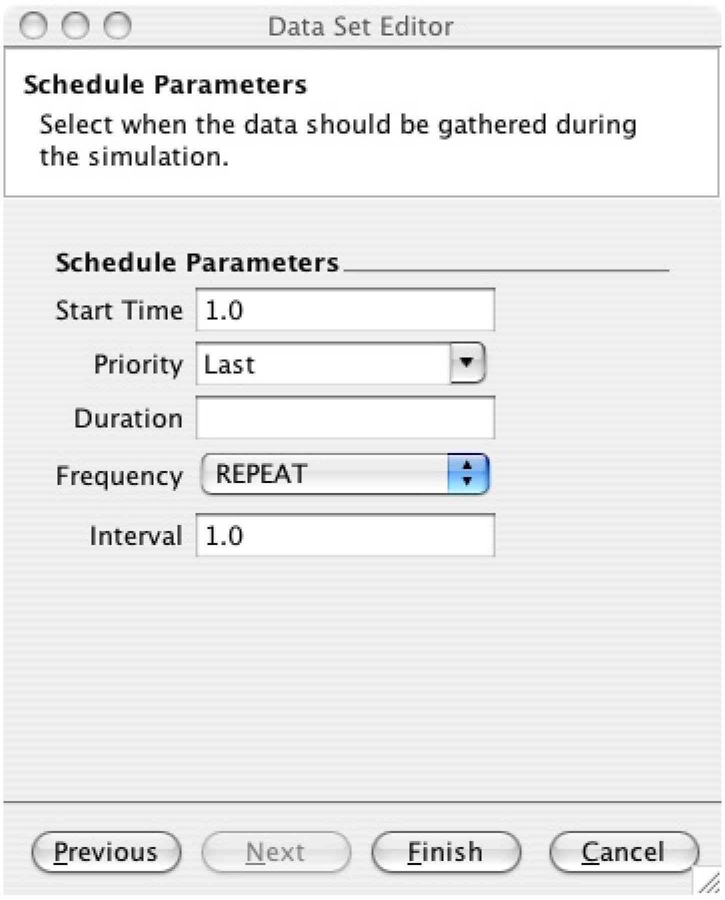
Assuming this new data set is used to record this data to a file and there are two agents named "John" and "Sam", the result would be something like:

```
"Tick", "Avg. Happiness"

1.0, 3.0
2.0, 1.5
```

The point being that the data is some aggregate value over all the agents.

Although it is possible to define aggregate and non-aggregate data sources in the same data set, its a good idea to keep them separate.

Step 3



Schedule Parameters

Select when the data should be gathered during the simulation.

Schedule Parameters

Start Time	1.0
Priority	Last
Duration	
Frequency	REPEAT
Interval	1.0

In this step, you schedule when you want the data to be collected.

Scripting Formulas

This page last changed on Aug 06, 2007 by collier.

Data values collected by a repast simphony simulation can be the result of a formula evaluation. The formulas are created using the Data Set wizard in the Formula Script option. This page describes the script's syntax and usage.

Getting Data from Agents

Formulas typically operate on data provided by the agents in the simulation. To retrieve data from an agent, the formula script uses a method call syntax. For example,

```
getWealth()
```

will call the getWealth() method on an agent or agents and store that value. If the script is operating in aggregate mode, the method will be called on every agent and the individual results will be placed into an array of double values. Subsequent operations will then operate on this array of values. If the script is operating in non-aggregate mode, the entire script will be evaluated against each agent, and thus a method call only occurs for a single agent. The result of the call, a single value, will then be placed into an array of size 1 and subsequent operations will operate on that array.

For example, in aggregate mode,

```
mean(getWealth())
```

will call getWealth on each agent and place the results in an array. The mean function will then calculate the mean value of all the values in the array and return that result. The same formula, in non-aggregate mode, would be evaluated for each agent, that is, the mean of the result of each call to get wealth would be returned. So, assuming there are 30 agents, the entire formula would be evaluated 30 times and result in 30 mean values. (Of course, using mean in a non-aggregate way doesn't make much sense.)

Note that the method calls must return a numeric **or** a boolean value. True is converted to 1.0 and the false is converted to 0.0

Functions and Operators

All the functions and operators operate in this array-based way. Listed in order of precedence, the functions and operators are:

1. abs, sqr, sqrt, exp, log10, log, sin, cos, tan, sind, cosd, tand, mean, sum, min, max, kurtosis, product, skewness, stddev, variance
2. ** (power)
3. /, *
4. +, -
5. <, <=, >, >=
6. ==, !=

The following constants are also supported.

1. E 2.7182818284590452354
2. PI 3.14159265358979323846

Functions

Functions operate on an array of double values such as those returned from a method call. A function may perform some operation on each element of an array and return a new array containing those results, or it may perform some operation across all the elements of the array and return a new array with a single value. For example, assuming aggregate mode and 10 agents, the following formula

```
sqrt(getWealth())
```

will take the square root of each of the 10 elements in the array produced by the getWealth() call and return a new 10 element array containing those square roots. In contrast,

```
sum(getWealth())
```

will sum all the 10 elements together and return the result as a single value.

The following functions operate on the array elements individually and return a new array equal to the size of input array.

- abs - returns the absolute value of each element in the input array
- sqrt - returns the square root of each element in the input array
- sqr - returns the square of each element in the input array
- log10 - returns the base 10 logarithm of each element in the input array
- exp - returns Euler's number raised the power of each element in the input array
- log - returns the natural logarithm of each element in the input array
- sin - returns the sine of each element in the input array. The elements are in radians
- cos - returns the cosine of each element in the input array. The elements are in radians
- tan - returns the tangent of each element in the input array. The elements are in radians
- sind - returns the sine of each element in the input array. The elements are in degrees.
- cosd - returns the cosine of each element in the input array. The elements are in degrees
- tand - returns the tangent of each element in the input array. The elements are in degrees

The following functions will return a single number (an array of size 1) in all cases.

- mean - returns the arithmetic mean of the all the elements in the input array
- sum - returns the sum of all the elements in the input array
- min - returns the minimum value of all the elements in the input array
- max - returns the maximum value of all the elements in the input array
- kurtosis - returns the kurtosis of all the elements in the input array. The kurtosis =
  Unknown macro: { [n(n+1) / (n -1)(n - 2)(n-3)] sum[(x_i - mean)^4] / std^4 }

  - [3(n-1)^2 / (n-2)(n-3)] where n is the number of values, mean is the mean and std is the standard deviation. Note that this statistic is undefined for n < 4. Double.Nan is returned when there is not sufficient data to compute the statistic.

- product - returns the product of all the elements in the input array.
- skewness - returns the skewness (unbiased) of all the elements in the input array. Skewness = [n / (n -1) (n - 2)] sum[(x_i - mean)^3] / std^3 where n is the number of values, mean in the mean and std is the is standard deviation.
- stddev - returns the unbiased "sample standard deviation" (the square root of the bias-corrected "sample variance") of all the elements in the input array
- variance - returns the unbiased "sample variance" of the all the elements in the input array. Variance = sum((x_i - mean)^2) / (n - 1) where mean is the mean and n is the number of elements

Arithmetic and Boolean operators

The arithmetic and boolean comparison operators operate are binary operators that operate on their left and right operands. These operands can be arrays of the **same** size, an array of any size and a single number or two numbers. They will return a an array of the same size as the input array. For example, in aggregate mode, when using an array and a single number such as,

```
getWealth() * 3
```

the operation is applied to each element of the array using the single number. So, the above will multiply each element of the array produced by the aggregate call to getWealth() by 3 and return a new array containing those results. When working with two arrays, the operation will apply to the corresponding element of each array. For example,

```
getWealth() - getExpenses()
```

will subtract the getExpenses() array element from the corresponding getWealth() array element and return a new array containing those results. That is, for all n: result[n] = getWealth[n] - getExpenses[n]. **Note** that when performing a binary operation on two arrays the arrays **must** be of the **same** size.

The arithmetic operators:

- +   returns the sum of the operands
- -   returns the difference of the operands
- *   returns the product of the operands
- /   returns the ratio of the operands
- **   returns the left operand raised to the power of the right operand.

The boolean binary operators perform the appropriate operation and return an array whose elements are the results of the operation. If the result is *true* then 1.0 is placed in the array, and if the result is *false* then 0.0 is placed in the array. As with the arithmetic operators, the boolean operators work on arrays of the same size, an array and a number or two numbers. For example,

```
getWealth() > 123
```

will return an array of the same size as that produced by the aggregated getWealth() call. This resulting array will contain 1.0 where the corresponding element in the getWealth() array is greater than 123 otherwise it will contain 0.0. That is, for all n: result[n] = getWealth[n] > 123 ? 1.0 : 0.0.

Two arrays can be compared in the same way. For example,

```
getWealth() > getExpenses()
```

will return an array whose elements contains 1.0 where the wealth element was greater than corresponding getExpenses element.

The boolean binary operators are:

- < returns 1 if the left operand is less than the right operand, else 0
- <= returns 1 if the left operand is less than or equal to the right operand, else 0
- > returns 1 if the left operand is greater than the right operand, else 0
- >= returns 1 if the left operand is greater than or equal to the right operand, else 0
- != returns 1 if the left operand is not equal to the right operand, else 0
- == returns 1 if the left operand is equal to the right operand, else 0

The boolean operators can be combined with the *sum* function to yield a count of those agents that passed the condition specified in the boolean expression. For example,

```
sum(getWealth() > 10)
```

will return a count of all the agents whose wealth is greater than 10.

The Final Result

The data set data collection mechanism expects to receive a single result when evaluating a formula. Consequently, only the first array element of the final array is recorded when a formula is evaluated. In the case of the aggregate function that produce a single result (i.e. *sum*, *mean* and so forth) this works as expected: the result of the function is recorded. Consequently, nearly all formulas when working in aggregate mode will use one these functions to produce a final result.

Watcher Queries

This page last changed on Nov 30, 2007 by collier.

Watcher queries are boolean expressions that evaluate the watcher and the watchee with respect to each other and some projection or context. The context is the context where the watcher resides and the projections are those contained by that context. In the following "[arg]" indicates that the arg is optional.

- colocated - true if the watcher and the watchee are in the same context.
- linked_to ['network name'] - true if the watcher is linked to the watchee in any network, or optionally in the named network
- linked_from ['network name'] - true if the watcher is linked from the watchee in any network, or optionally in the named network
- within X ['network name'] - true if the path from the watcher to the watchee is less than or equal to X where X is a double precision number. This is either for any network in the context or in the named network.
- within_vn X ['grid name'] - true if the watchee is in the watcher's von neumann neighborhood in any grid projection or in the named grid. X is the extent of the neighborhood in the x, y, [z] dimensions.

- within_moore X ['grid name'] - true if the watchee is in the watcher's moore neighborhood in any grid projection or in the named grid. X is the extent of the neighborhood in the x, y, [z] dimensions.
- within X ['geography name'] true if the orthodromic distance from the watcher to the watchee is less than or equal to X meters, otherwise false.

Watcher queries can be combined using the keywords ***not*** and ***and*** as well as ***(*** to establish precedence. For example,

within 2 'family' and not linked_to 'business'

The queries are tested in WatcherQueryTests and defined as annotations in MyWatcher.

Working with the Scheduler

This page last changed on Nov 22, 2010 by north.

There are basically three ways to work with the Repast Simphony Scheduler. No one is better than the other and they each have their specified purpose.

Directly Schedule an action

This is similar to the way that actions have always been scheduled in repast with a slight twist. In this scenario, you get a schedule and tell it the when and what to run. An example of adding an item to the schedule this way is as follows:

```
//Specify that the action should start at tick 1 and execute every other tick
ScheduleParameters params = ScheduleParameters.createRepeating(1, 2);

//Schedule my agent to execute the move method given the specified schedule parameters.&nbsp;
schedule.schedule(params, myAgent, "move");
```

The biggest change here is that instead of using one of the numerous methods to set up the parameters for the action like you would in repast 3, you just use an instance of ScheduleParameters. ScheduleParameters can be created using one of several convient factory methods if desired. A slight alteration of this is:

```
//Specify that the action should start at tick 1 and execute every other tick
ScheduleParameters params = ScheduleParameters.createRepeating(1, 2);
//Schedule my agent to execute the move method given the specified schedule parameters.
schedule.schedule(params, myAgent, "move", "Forward", 4);
```

This example schedules the same action with the same parameters, but passes arguments to the method that is to be called. The "Forward" and 4 will be passed as arguments to the method move. The assumption is that the signature for move looks like this:

```
public void move(String direction, int distance)
```

Schedule with Annotations

Java 5 introduced several new and exciting features (some of which are used above), but one of the most useful is Annotation support. Annotations, in java, are bits of metadata which can be attached to classes, methods or fields that are available at runtime to give the system more information. Notable uses outside repast includes the EJB3 spec which allows you to create ejbs using annotations without requiring such complex descriptors. For repast, we thought annotations were a perfect match for tying certain types of scheduling information to the methods that should be scheduled. The typical case where you would use annotations is where you have actions whose schedule is know at compile time. So for example, if you know that you want to have the paper delivered every morning, it would be logical to schedule the deliverPaper() method using annotations. Without going into extensive documentation about how annotations work (if you want that look at Java 5 Annotations), here is how you would schedule an action using annotations:

```
@ScheduledMethod(start=1 , interval=2)
public void deliverPaper()
```

The arguments of the annotation are similar to the properties for the ScheduleParameters object. One particularly nice feature of using annotations for scheduling is that you get keep the schedule information right next to the method that is scheduled, so it is easy to keep track of what is executing when.

Most of the time, objects with annotations will automatically be added to the schedule, however, if you create a new object while your simulation is running, this may not be the case. Fortunately, the schedule object makes it very easy to schedule objects with annoations.

```
//Add the annotated methods from the agent to the schedule.
schedule.schedule(myAgent);
```

The schedule will search the object for any methods which have annotations and add those methods to the schedule.

This type of scheduling is not designed to handle *dynamic* scheduling, but only scheduling, where the actions are well defined at compile time.

Schedule with Watcher

Scheduling using watchers is the most radical of the new scheduling approaches. Watchers are designed to be used for dynamic scheduling where a typical workflow is well understood by the model designer. Basically, a watcher allows an agent to be notified of a state change in another agent and schedule an event to occur as a result. The watcher is set up using an annotation (like above), but instead of using static times for the schedule parameters, the user specifies a query defining whom to watch and a query defining a trigger condition that must be met to execute the action. That's a bit of a mouthfull, so let's take a look at an example to hopefully clarify how this works. (this code is from the SimpleHappyAgent model which ships with Repast Simphony)

```
@Watch(watcheeClassName = "repast.demo.simple.SimpleHappyAgent", watcheeFieldName =
"happiness", query = "linked_from",
                whenToTrigger = WatcherTriggerSchedule.LATER, scheduleTriggerDelta = 1,
scheduleTriggerPriority = 0)
public void friendChanged(SimpleHappyAgent friend) {
    if (Math.random() > .25) {
        this.setHappiness(friend.getHappiness());
    } else {
        this.setHappiness(Random.uniform.nextDouble());
    }
    System.out.println("Happiness Changed");
}
```

There is a fair amount going on in this, so we'll parse it out piece by piece (for a more detailed techical explanation, see [[fill in link]]).

First, note that there is a @Watch annotation before the method. This tells the Repast Simphony system that this is going to be watching other objects in order to schedule the friendChanged() method. The first parameter of the annotation is the watcheeClassName. This defines the type of agents that this object will be watching. The second argument, watcheeFieldName, defines what field we are interested in monitoring. This means that there is a variable in the class SimpleHappyAgent, that we want to monitor for changes. When it changes, this object will be notified. The query argument defines which instances of SimpleHappyAgent we want to monitor. In this case we are monitoring agents to whom we are linked. For more documentation on arguments for this query can be found at: Watcher Queries. The whenToTrigger argument specifies whether to execute the action immediately (before the other actions at this time are executed) or to wait until the next tick. The scheduleTriggerDelta defines how long to wait before scheduling the action (in ticks). Finally the scheduleTriggerPriority allows you to help define the order in which this action is executed at it's scheduled time.

Let me give a practical example of how this would be used. Let's say you are modelling electrical networks. You may want to say that if a transformer shuts down, at some time in the future a plant will shut down. So you make the plant a watcher of the transformer. The plant could watch a variable called status on the transformers to which it is connected, and when the transformer's status changes to OFF, then the plant can schedule to shut down in the future. All done with a single annotation. It could look like this:

```
@Watch(watcheeClassName = "infr.Transformer", watcheeFieldName = "status", query =
"linked_from",
                whenToTrigger = WatcherTriggerSchedule.LATER, scheduleTriggerDelta = 1)
public void shutDown(){
    operational = false;
}
```

Obviously that is a simple example, but it should give you an idea of how to work with scheduling this way.