Analysis of MATSim

Table of Contents

Background

Multi-Agent Transport Simulation Toolkit (MATSim) simulations, with maps and people on the maps, seem like the type of spatial simulation the MASS library is well suited to handle. A MATSim simulation using MASS could be compared with other MATSim simulations to measure the performance of MASS.

The MATSim project is at http://matsim.org/. It uses Maven (http://maven.apache.org/) for its dependency and build management. Maven is a command-line tool, but there are plugins for both the Netbeans and the Eclipse IDEs. The MATSim project group has set up MATSim as an Eclipse project, but using a different IDE should not be a problem, as Eclipse itself is not required.

Brief MATSim Execution Overview

MATSim has a main() method in org.matsim.run.Controler, which is a wrapper class for org.matsim.core.Controler, which handles the running of the simulation. The main method instantiates a Controler and calls run().

In the Controler (org.matsim.core), data about the simulation is read in from an XML configuration file. The configuration file is organized by modules, denoted by <module> tags. The Controler stores the

module information in a Scenario object. Several modules are built in to MATSim, and it's also possible to create modules of one's own. Custom modules seem to be used to load objects that can be used in the simulation by registering them with the Controler as event listeners. Figure 1 below shows a simple configuration file that comes with MATSim.

```
<?xml version="1.0" ?>
<!DOCTYPE config SYSTEM "http://www.matsim.org/files/dtd/config_v1.dtd">
<!-- Fairly minimalistic config file.  See config dump in console or in log file for more options, and some
explanations. -->
<config>

	<module name="network">
		<param name="inputNetworkFile" value="examples/equil/network.xml" />
	</module>

	<module name="plans">
		<param name="inputPlansFile" value="examples/equil/plans100.xml" />
	</module>

	<module name="controler">
		<param name="outputDirectory" value="./output/example1" />
		<param name="eventsFileFormat" value="xml" />
		<param name="firstIteration" value="0" />
		<param name="lastIteration" value="0" />
		<param name="mobsim" value="queueSimulation" />
	</module>

	<module name="planCalcScore" >
		<param name="activityType_0" value="h" />
		<param name="activityType_1" value="w" />
		<param name="activityTypicalDuration_0" value="12:00:00" />
		<param name="activityTypicalDuration_1" value="08:00:00" />
	</module>


</config>
```

Figure 1. Simple configuration file at matsim/examples/tutorial/config/example1-config.xml

A MATSim map is called a network. In figure 1, the "network" module indicates the location of the XML file holding the network information. The people in a MATSim simulation are called the population. Information about the population is held in a "plans" file. The "plans" module points indicates the location of the plans file.

In Figure 1, notice the "mobsim" parameter in the "controler" module. The parameter's value is "queueSimulation", which refers to the basic simulation type built in to MATSim. There are three other types of simulations built in to MATSim that all seem to be enhanced versions of the Queue Simulation, the JDEQ Simulation, the QSim Simulation, and the MultiModalMobsim Simulation.

A network is represented in the configuration with nodes and links. Figure 2 shows XML snippets showing how their representation.

```
<nodes>
      <node id="1" x="-20000" y="0"/>
      <node id="2" x="-15000" y="0"/>
…
<links capperiod="01:00:00">
      <link id="1" from="1" to="2" length="10000.00" capacity="36000" freespeed="27.78" permlanes="1"  />
      <link id="2" from="2" to="3" length="10000.00" capacity="3600" freespeed="27.78" permlanes="1"  />
      <link id="3" from="2" to="4" length="10000.00" capacity="3600" freespeed="27.78" permlanes="1"  />
…
```

Figure 2. Nodes and links in XML representation.

The population is represented with <person> elements in the XML file for the "plans" module. Figure 3 shows an XML snippet representing a person.

```
<person id="2">
        <plan>
                <act type="h" x="-25000" y="0" link="1" end_time="05:59" />
                <leg mode="car">
                        <route>2 7 12</route>
                </leg>
                <act type="w" x="10000" y="0" link="20" dur="02:30" />
                <leg mode="car">
                        <route>13 14 15 1</route>
                </leg>
                <act type="h" x="-25000" y="0" link="1" />
        </plan>
</person>
```

Figure 3. A person in a simulation population in XML representation.

Once the Controler loads the simulation configuration and module information, it runs the simulation. MATSim is designed to run a simulation multiple times, so a simulation runs through a number of iterations, as specified in the XML configuration file. The Controler uses the Simulation interface (org.matsim.core.mobsim.framework.Simulation) to do the running of the simulation. The Simulation interface has only a run() method. The implementation of the Simulation varies depending on the simulation indicated in the XML configuration file.

As the simulation runs, each person in the population wends its way through a plan consisting of activities and legs. The plan for the person in figure 3 can be seen in the XML snippet. Information about each person's progress through the network is logged for interpretation and study.

MATSim API

MATSim has an API in the package org.matsim.api.core.v01 and its subpackages, as shown in the snapshot in figure 4.

Figure 4. API packages highlighted in green.

Figure 5 shows some of the org.matsim.api.core.v01.network package.

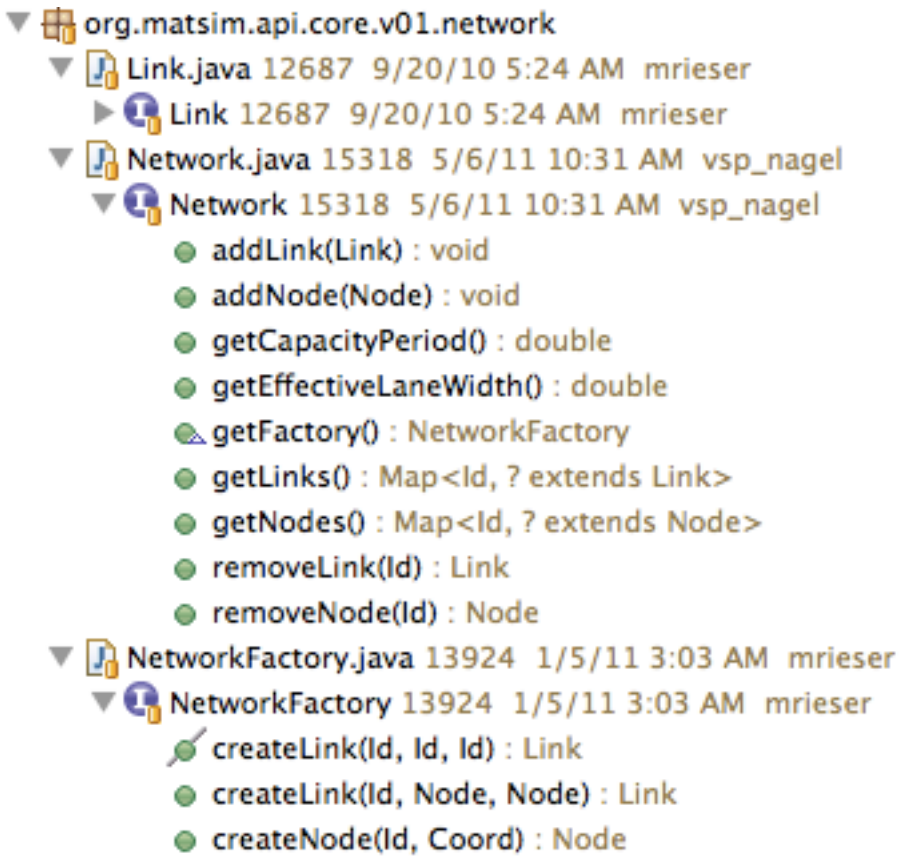

Figure 5. View of part of the network interface.

Implementations and Factories

A convention of the MATSim project is to name implementations of the interfaces with "Impl" after the interface name. For example, the Network interface is implemented with the NetworkImpl class. Classes are often coupled with factory classes. The factory classes normally implement an interface with "Factory" at the end of the name. The factory classes follow the convention of appending "Impl" to the factory's interfact name. For example, the NetworkImpl class works with the NetworkFactoryImpl class to create the nodes and links of a simulation's network. Likewise, the Network interface has a corresponding NetworkFactory interface, as shown in figure 5 above.

Porting MATSim to MASS

There are three important areas to consider in porting MATSim to MASS: the network, the population, and the simulation. The MATSim network will be implemented with a MASS.Places object. The population will be implemented with a MASS.Agents object. And the simulation will manage the Places and Agents through MASS calls.

The MATSim Network

MATSim automatically creates a core network (org.matsim.core.network.NetworkImpl) that can be used to create a specialized network for a particular simulation. The core network is created when the configuration files are parsed. The core network has three important data structures, a Map of links arranged by ID number, a Map of nodes also arranged by ID number, and a QuadTree of nodes arranged by geographic location.

The QuadTree is a useful structure for organizing items by location on a two-dimensional space. The concept behind it is to take a square area with boundaries in a specific square (or rectangle). The square is the root node. The root node is filled like a bucket with leaves defined by coordinates. As the root node fills to a predefined capacity, it creates four nodes, each representing a quarter of its square, and apportions the leaves among the new nodes according to which square contains the coordinates of a given leaf. Each of the new nodes is filled like a bucket until it fills to capacity and divides into quarters and spawns four nodes of its own. The result is a tree with leaves sorted by geographic location. Figure 6 illustrates the squares and tree.

The QuadTree is of particular interest for MASS because it seems ideal to organize a set of Place objects so that the Place objects geographically close to one another in the network are also close to one another in the Places array, hopefully generating as much locality as possible in the Places data structure.

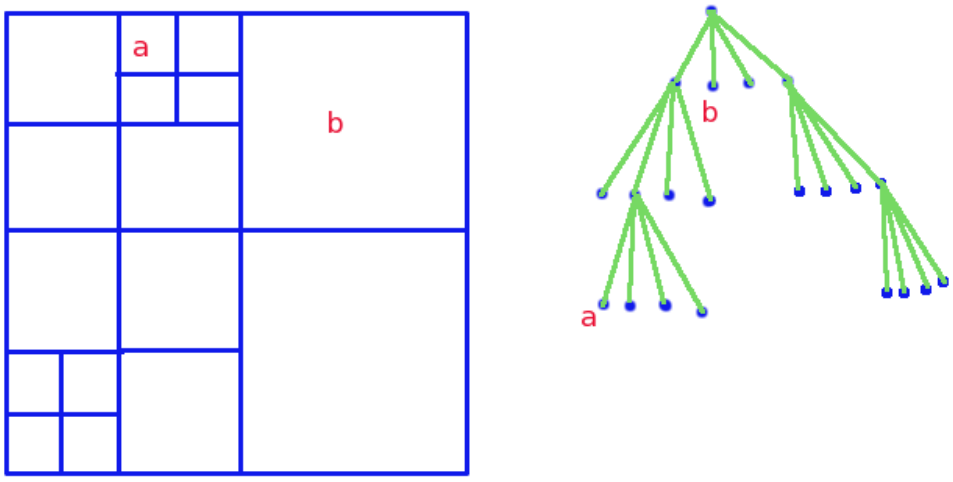

Figure 6 Diagrams illustrating a square and tree representation of a QuadTree.

The QuadTree is

The specialized network is created later by the simulation, before it begins to run through its iterations. The Queue Simulation has its own network class and network factory, QueueNetwork and QueueNetworkFactory. When the Queue Simulation is instantiated by the Controler, QueueSimulation accesses the core network through a ScenarioImpl class. The ScenarioImpl is provided by the Controler class and holds all the basic information read in from the various xml files on startup. Once in touch with the QueueNetwork, QueueSimulation creates its QueueNetwork with the aid of a QueueNetworkFactory. Figure 7 shows the main features of this.

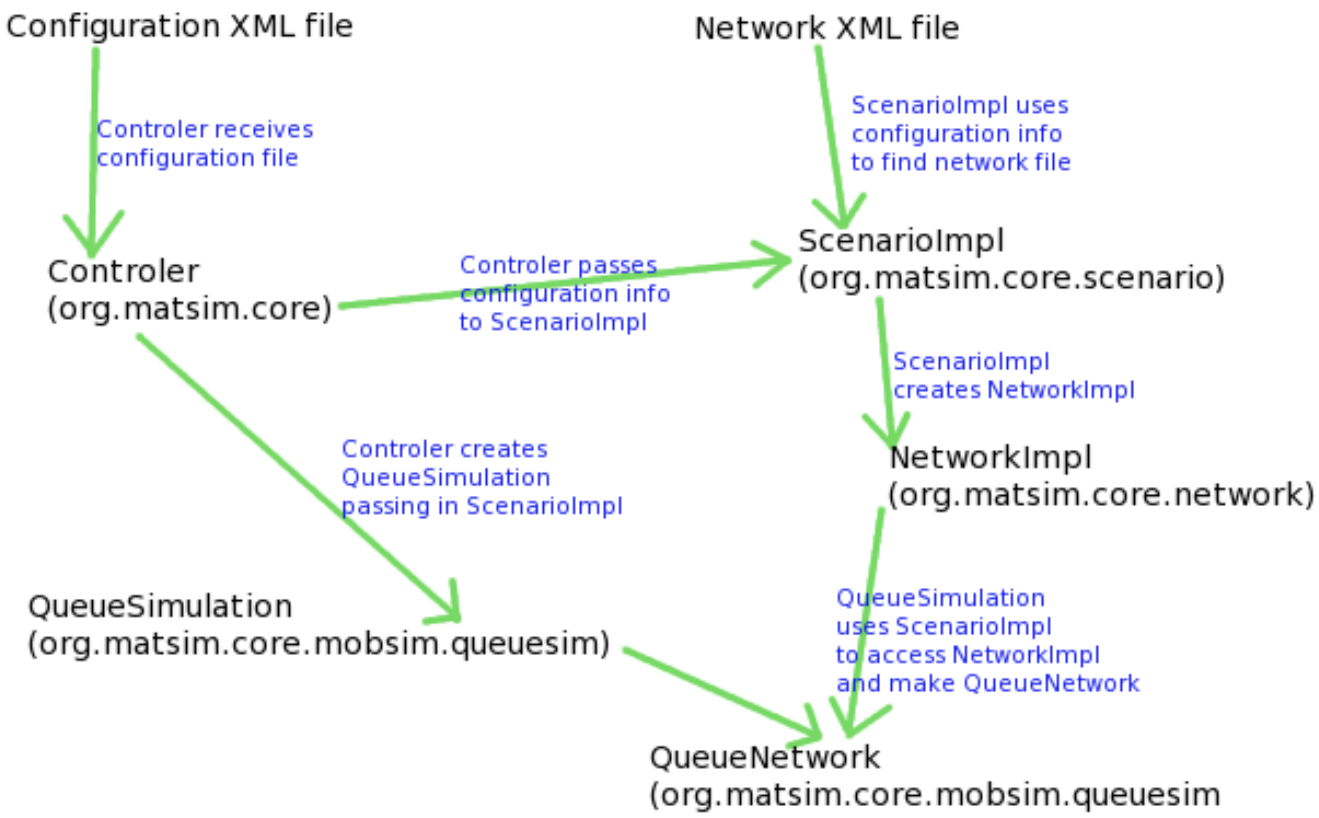

Figure 7. Diagram of the main steps to creation of a specialized network for the QueueSimulation.

MASS Network Implementation Starting Point

A class to replace QueueNetwork would be useful for a MASS simulation, say the MASSNetwork class. Along with this, a MASSNetworkFactory would be helpful for setting up the MASSNetwork.

Main duties for the MASSNetworkFactory would be as follows:

- Start the MASS environment
- Examine the NetworkImpl to determine the correct size for a Places object
- Instantiate the Places object and return it to the simulation to be used as a MASSNetwork.

The main duty of the MASSNetwork would be to serve as an entry point for a MASS simulation to execute MASS calls to interact with the map, as described below.

The QueueSimulation sets up the QueueNetwork in its constructor. A similar, early point would seem a good place for a MASS simulation to create a MASSNetworkFactory and MASSNetwork.

MATSim Population

The QueueSimulation uses the core MATSim Population class, PopulationImpl (org.matsim.core.population), as provided by the ScenarioImpl, to create its own representation of the population. It does this in the createAgents() method of QueueSimulation. QueueSimulation takes the individual persons from the core population and couples them with vehicles. For this it uses the AgentFactory class of the in the QueueSimulation package (org.matsim.core.mobsim.queuesim). The AgentFactory is used in the createAgents() method of QueueSimulation, which is called shortly after the

QueueSimulation's run() method. After modifying the population to support vehicles, QueueSimulation keeps its population in an ArrayList.

MASS Population Implementation Starting Point

Again, as with the network, a similar setup can be used for a MASS simulation. Using the core population, a MASSPopulationFactory could check the core population to determine a correct size for an Agents object and instantiate it. This agents object would be the entry point for the simulation to make MASS calls to interact with the agents, as described below.

Simulation

For a MATSim simulation based on MASS, a simulation class, say MASSSimulation, will be needed. Main responsibilities of the MASSSimulation class would be to make MASS calls at each step of the simulation to advance or manage the agents through the MASSNetwork.

Various approaches can be used to advance the simulation. The QueueSimulation makes steps in increments of time. Another built in simulation, JDEQSimulation, is event-based, triggering events as agents calculate they will be ready for them. A MASS Simulation could take either approach.

API Beneath the Population and Network

An implementation of a MASS-based network and population should interface with the network and population elements the network locations and population agents will have to manage. For example, a MATSim agent must interact with classes used to manage its progress and determine arrivals. These are standardized in the API in org.matsim.api.core.v01, mentioned above. It is important that a MASS-based simulation provide agents and places that can integrate with these interfaces and make use of existing classes.

Appendix: A MASS-Thread Application With Places and Agents

A MASS implementation of a MATSim simulation will require a MASS environment with agents. This Appendix contains the source code for a multi-threaded MASS application called SimpleTest that incorporates agents. The source code for the MASS library implementation is available in the ~dslab/…/MASS-Thread folder. The MASS-Thread code is too voluminous to include here.

SimpleTest

The SimpleTest application creates a two-dimensional Places object and an Agents object to populate it. The Territory class extends Place, and the Troop class extends Agent. In the SimpleTest window, each Territory is shown as a rectangular area outlined in red. Troops are shown as filled circles. A timer updates the position of each Troops within its Territory. Buttons cause Troops to randomly sleep, schedule random wakeupAll events, schedule random migrate events, and trigger manageAll calls. When Troops sleep they stop moving, and when they wake up they start again. Figure 8 shows a screenshot of the demonstration.

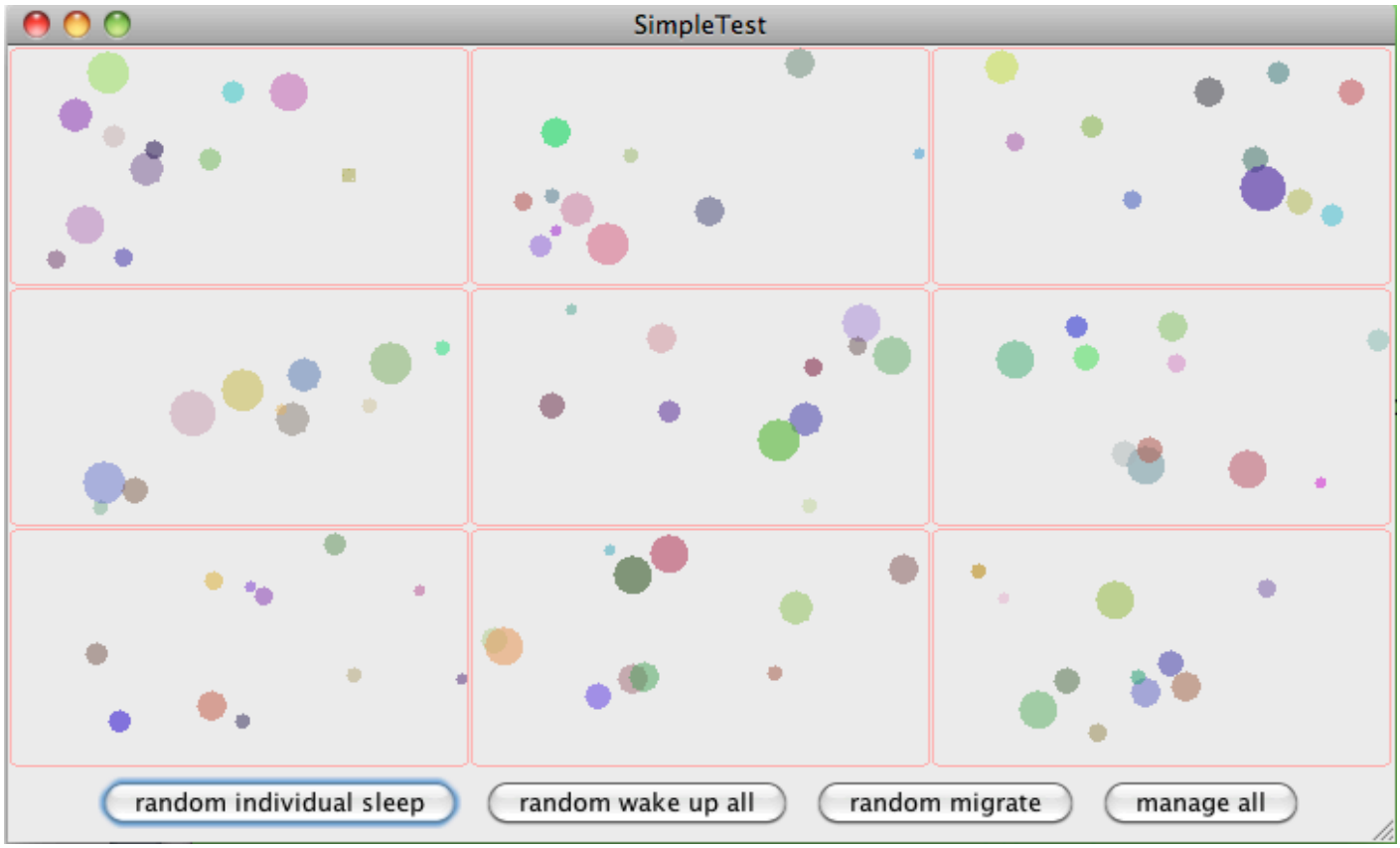

Figure 8. Screenshot of the SimpleTest application

SimpleTest Source Code

There are five files in the SimpleTest source code:

- SimpleTest.java – contains the main() method
- Territory.java – extends Place
- Troop.java – extends Agent
- PaintBundle.java – a helper class for collating data
- TroopBundle.java – a helper class for collating data

SimpleTest.java

```
package SimpleTest;

import MASS.*;
import java.util.Vector;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/* A class for performing a simple test. Started with
 * the Mass.Template.java file in the dslab MASS folder.
 *
 * @author John Spiger
 * @created 5/17/11
 */
public class SimpleTest extends JPanel implements ActionListener {

  Places territories;
```

```
Agents troops;
JPanel controls;
JButton randomSleep;
JButton randomWakeUpAll;
JButton randomMigrate;
JButton manageAll;

private SimpleTest () throws Exception {
    setPreferredSize(new Dimension(800, 800));
    territories = new Places( 1, "SimpleTest.Territory", null, 4, 4);
    troops = new Agents( 2, "SimpleTest.Troop", null, territories, 100 );
}

private JPanel getControls(){
    controls = new JPanel();
    randomSleep = new JButton("random individual sleep");
    randomSleep.addActionListener(this);
    controls.add(randomSleep);
    randomWakeUpAll = new JButton("random wake up all");
    randomWakeUpAll.addActionListener(this);
    controls.add(randomWakeUpAll);
    randomMigrate = new JButton("random migrate");
    randomMigrate.addActionListener(this);
    controls.add(randomMigrate);
    manageAll = new JButton("manage all");
    manageAll.addActionListener(this);
    controls.add(manageAll);
    return controls;
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == randomSleep){
      troops.callAll(Troop.RANDOM_SLEEP);
    }
    if (e.getSource() == manageAll){
      troops.manageAll();
    }
    if (e.getSource() == randomWakeUpAll){
      troops.callAll(Troop.RANDOM_WAKEUPALL);
    }
    if (e.getSource() == randomMigrate){
      troops.callAll(Troop.MIGRATE);
    }
}

private void runSimulation(){
    boolean threadOkay = true;
    int counter = 0;
    int counter1 = 0;
    int counter2 = 0;
    while (threadOkay){
      troops.callAll(Troop.UPDATE, new Long(System.currentTimeMillis()));
      repaint();
      try {
     Thread.currentThread().sleep(50); //20fps
      } catch (Exception e){
     threadOkay = false;
```

```
        }
      }
  }

public void paint(Graphics g){
      super.paint(g);
      PaintBundle pb = new PaintBundle(g, getSize());
      territories.callAll(Territory.PAINT, pb);
      troops.callAll(Troop.PAINT, g);
  }
  /**
   * Create the GUI and show it.  For thread safety,
   * this method should be invoked from the
   * event-dispatching thread.
   */
  private static void createAndShowGUI( SimpleTest simpleTest) {
      try {
        //Create and set up the window.
        JFrame frame = new JFrame("SimpleTest");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        //Create and set up the content pane.
        frame.add(simpleTest, BorderLayout.CENTER);
        frame.add(simpleTest.getControls(), BorderLayout.SOUTH);
        //Display the window.
        frame.pack();
        frame.setVisible(true);
      } catch (Exception e){
        e.printStackTrace();
      }
  }

  public static void main(String[] args) {
      try{
        MASS.init( args );
        final SimpleTest simpleTest = new SimpleTest();
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
       public void run() {
         createAndShowGUI( simpleTest );
       }
        });
        simpleTest.runSimulation();
      } catch (Exception e){
        System.out.println("ERROR:");
        e.printStackTrace();
      }
  }
}
```

Territory.java

```
/*
 * A test class for use with SimpleTest, started
 * with sample code in MassTemplate.java from the dslab folder.
 */
```

```
package SimpleTest;

import MASS.*;
import java.util.*;
import java.awt.*;

public class Territory extends Place {

  private Rectangle rectangle;

  public static final int MUSTER = 1;
  public static final int PAINT = 2;

  public Territory( Object args ) {
      super();
  }

  // -------------------------------------------------------
  private String getIndexString(){
      String str = "";
      for (int i = 0; i < index.length; i++){
        str += index[i];
        if (i < index.length -1){ str += " "; }
      }
      return str;
  }

  // -------------------------------------------------------------

  /*
   * @return the Rectangle this Territory calculated for itself
   * on the last call to PAINT, null if none yet calculated
   */
  public Rectangle getRectangle(){
      return rectangle;
  }
  // -------------------------------------------------------------

  public Object callMethod( int functionId, Object argument){
      Object retVal = null;
      switch (functionId) {
        case MUSTER:
       retVal = _muster(argument);
        case PAINT:
       _paint(argument);
       break;
        default:
       break;
      }
      return retVal;
  }

  // -------------------------------------------------------------

  private void _paint(Object argument){
      if (argument instanceof PaintBundle){
        PaintBundle gb = (PaintBundle)argument;
```

```
        Dimension d = gb.dimension;
        Graphics g = gb.graphics;
        //find dims for this territory
        int panelWidth = d.width;
        int panelHeight = d.height;
        int xIndex = 0;
        int yIndex = 0;
        int xSize = 1;
        int ySize = 1;
        if (this.index.length == 2 && this.size.length == 2){
      xIndex = this.index[0];
      yIndex = this.index[1];
      xSize = this.size[0];
      ySize = this.size[1];
        }
        int rectWidth = panelWidth/xSize ;
        int rectHeight = panelHeight/ySize;
        int rectX = rectWidth * xIndex;
        int rectY = rectHeight * yIndex;
        //put in some border space
        if (rectWidth > 10 && rectHeight > 10){
      rectWidth -= 2;
      rectX++;
      rectHeight -= 2;
      rectY++;
        }
        rectangle = new Rectangle(rectX, rectY, rectWidth, rectHeight);
        g.setColor(Color.pink);
        g.drawRoundRect(rectX, rectY, rectWidth, rectHeight
                        , rectWidth/20, rectHeight/20);
    }
  }

  // ---------------------------------------------------------------

  private Object _muster(Object o){

      return "muster";
  }
}
```

PaintBundle.java

```
package SimpleTest;

import java.awt.Graphics;
import java.awt.Dimension;

/*
 * A class for bundling up information an object needs in order to know where to paint;
 */
class PaintBundle {
  public Graphics graphics;
  public Dimension dimension;

  public PaintBundle(Graphics g, Dimension d){
      graphics = g;
```

```
      dimension = d;
  }
}
```

Troop.java

```
/*
 * A test class for use with SimpleTest, started
 * with sample code in MassTemplate.java from the dslab folder.
 */
package SimpleTest;
import MASS.*;
import java.awt.*;

public class Troop extends Agent {
  public static final int REPORT = 0;
  public static final int THROW_DART_ON_MAP = 1;
  public static final int WINK = 2;
  public static final int PAINT = 3;
  public static final int UPDATE = 4;
  public static final int RANDOM_SLEEP = 5;
  public static final int RANDOM_WAKEUPALL = 7;
  public static final int MIGRATE = 8;


  private double xRatio;
  private double yRatio;
  private double radiusRatio;
  private double direction; //radians
  private double speed; // percentage of width and/or height across Place per second
  private double baseSpeed;
  private long lastUpdateTime;
  private Color color;
  private Color baseColor;
  private boolean sleeping;
  private boolean migrating;

  private static Color paleRed = new Color(1.0f, 0.0f, 0.0f);//, 0.25f);
  private static Color paleBlue = new Color(0.0f, 0.0f, 1.0f);//, 0.25f);
  private static Color paleGreen = new Color(0.0f, 1.0f, 0.0f);//, 0.25f);
  private static Color paleBlack = new Color(0.0f, 0.0f, 0.0f);//, 0.25f);

  public Troop( Object args ) {
      super();
      TroopBundle tb;
      if (args instanceof TroopBundle){
        tb = (TroopBundle)args;
      } else {
        tb = new TroopBundle();
      }
      color = tb.color;
      baseColor = tb.color;
      double dRat = tb.directionRatio;
      direction = (dRat >= 0.0 && dRat <= 1.0) ? dRat * Math.PI * 2 : 0.0 ;
      double radiusScale = 0.075;
      double radiusBump = 0.025;
      radiusRatio = (tb.radiusRatio >= 0.0 && tb.radiusRatio <= 1.0) ? tb.radiusRatio * radiusScale
```

```
: 0.2 * radiusScale;
      radiusRatio += radiusBump;
      xRatio = (tb.xRatio >= 0.0 && tb.xRatio <= 1.0) ? tb.xRatio : 0.5;
      yRatio = (tb.yRatio >= 0.0 && tb.yRatio <= 1.0) ? tb.yRatio : 0.5;
      double throttle = 4.0;
      speed = (tb.speedRatio >= 0.0 && tb.speedRatio <= 1.0) ? tb.speedRatio/throttle : .5 /
throttle ;
      speed += 0.2;
      baseSpeed = speed;
      sleeping = false;
      migrating = false;
      lastUpdateTime = System.currentTimeMillis();
  }

  public Object callMethod(int functionId, Object argument){
      Object retVal = null;
      switch (functionId) {
        case REPORT:
       retVal = _report(argument);
       break;
        case THROW_DART_ON_MAP:
       _throwDartOnMap(argument);
       break;
        case WINK:
       _wink(argument);
       break;
        case PAINT:
       _paint(argument);
       break;
        case UPDATE:
       _update(argument);
       break;
        case RANDOM_SLEEP:
       _randomSleep(argument);
       break;
        case RANDOM_WAKEUPALL:
       _randomWakeUpAll(argument);
       break;
        case MIGRATE:
       _migrate(argument);
       break;

        default:
       break;
      }
      return retVal;
  }
  // ----------------------------------------------------------------

  private void _migrate(Object argument){
      int roll = (int)(Math.random() * 10);
      if (roll < 3){
        int[] size = this.getPlace().size;
        int number = Places.indexArr2Num(this.getPlace().index.clone(), size);
        int max = 1;
        for (int i = 0; i < size.length; i++){
       max *= size[i];
```

```
      }
      roll = (int)(Math.random() * max);
      int[] newIndex = Places.indexNum2Arr(roll, this.getPlace().size);
      if (migrate(newIndex)){
     migrating = true;
      }
    }
}
// ------------------------------------------------------------------

private void _randomSleep(Object argument){
    int die = 4; //rolling for doubles
    int roll = (int)(Math.random() * die) + 1 ;
    int chance = (int)(Math.random() * die) + 1 ;
    if (chance == 1) {
      boolean slept = sleep(roll);
      if (slept){
     sleeping = true;
     speed = 0;
     switch (roll) {
       case 1:
            color = paleRed;
            break;
       case 2:
            color = paleGreen;
            break;
       case 3:
            color = paleBlue;
            break;
       default:
            color = paleBlack;
            break;
     }
      }
    }
}

private void _randomWakeUpAll(Object argument){
    int chance = 4;
    chance = (int)(Math.random() * chance);
    if (chance == 0){
      int eventId;
      if (argument instanceof Integer){
     eventId = ((Integer)argument).intValue();
      } else {
     eventId = (int)(Math.random() * 4) + 1;
      }
      wakeupAll(eventId);
    }
}

// ------------------------------------------------------------------

public void wakeup(int eventId){
      sleeping = false;
}
```

```
// ------------------------------------------------------------

private void _update(Object argument){
    if (!sleeping && speed < baseSpeed){
      speed += (baseSpeed/20.0d);
    }
    long currentTime;
    if (argument instanceof Long){
      currentTime = ((Long)argument).longValue();
    } else {
      currentTime = System.currentTimeMillis();
    }
    double lapse = ((double)(lastUpdateTime - currentTime))/1000; //seconds
    lastUpdateTime = currentTime; //update time
    double distance = lapse * speed;
    yRatio += distance * (Math.sin(direction));
    xRatio += distance * (Math.cos(direction));
    //if hitting an edge
    if (yRatio < 0.0 || yRatio > 1.0 || xRatio < 0.0 || xRatio > 1.0){
      double halfPi = Math.PI/2;
      double twoPi = Math.PI * 2;
      boolean high = (yRatio < 0.0) ? true : false;
      boolean low = (yRatio > 1.0) ? true : false;
      boolean left = (xRatio < 0.0) ? true : false;
      boolean right = (xRatio > 1.0) ? true : false;
      //determine new direction
      boolean south = (direction > Math.PI && direction != 0) ? true : false;
      boolean north = (direction > 0 && direction < Math.PI) ? true : false ;
      boolean west = (direction < halfPi || direction > (3 * halfPi)) ? true : false;
      boolean east = (direction > halfPi && direction < (3 * halfPi)) ? true : false;
      boolean corner = ((high && right)||(high && left)
                        ||(low && right)||(low && left)) ? true : false;
      if (corner
       || ((high || low ) && !(west || east))
       || ((right || left) && !(north || south))){
     direction = (direction + Math.PI)%(twoPi); //U-turn
      } else {
     double theta = (direction % (halfPi));
     boolean clockwise = (high && east
                                    || low && west
                                    || right && south
                                    || left && north) ? true : false;
     if (clockwise){ //decrease angle of direction
       direction = (direction + (Math.PI - 2 * theta)) % twoPi;
     } else { //increase angle of direction
       direction = (direction - (2 * theta)) % twoPi;
       if (direction < 0.0) direction += twoPi;
     }
      }
      //reset xRatio and yRatio
      if (left) xRatio = 0.0;
      if (right) xRatio = 1.0;
      if (high) yRatio = 0.0;
      if (low) yRatio = 1.0;
    }
}
```

```
// ------------------------------------------------------------

private void _paint(Object argument){
    Place p = getPlace();
    if (argument instanceof Graphics && p instanceof Territory){
      Graphics g = (Graphics)argument;
      Color originalColor = g.getColor();
      Territory t = (Territory)p;
      Rectangle r = t.getRectangle();
      if (r != null){
     int smallerDim = (r.width < r.height) ? r.width : r.height;
     int radius = (int)(radiusRatio * smallerDim);
     radius = (radius < 1) ? 1 : radius;
     int max = 100;
     radius = (radius > max) ? max : radius;
     int x = (int)((r.width - 2 * radius) * xRatio) + r.x;
     int y = (int)((r.height - 2 * radius) * yRatio) + r.y;
     g.setColor(color);
     g.fillOval(x, y, radius * 2, radius * 2);
     if (agentId == 0){
       g.drawRect(x,y, radius*2, radius*2);}
      }
      g.setColor(originalColor);
    }
}
// ------------------------------------------------------------

private void _wink(Object argument){
    System.out.println(this.agentId + " winks: " + (String)argument);
}

private Object _report(Object argument){
    String str = "troop #" + this.agentId + ", ";
    int[] index = this.getPlace().index;
    for (int i = 0; i < index.length; i++){
      str += " " + index[i];
    }
    return str;
}

private void _throwDartOnMap(Object o){
    int chance = 6; //one in 'chance' chance of migrating
    long placesLen = 1;
    int [] size = getPlace().size;
    for (int i = 0; i < size.length; i++){
      placesLen *= size[i];
    }
    int straw = (int)(System.nanoTime() % placesLen);
    if (straw == 0){ //this agen is a winner and will migrate
      int myIndexNum = Places.indexArr2Num( getPlace().index, size);
      placesLen--; //reduce by one to make sure same place won't be chosen
      straw = (int)(System.nanoTime() % placesLen);
      if (straw >= myIndexNum) {
     straw++; //make up for one taken from placesLen
      }
      migrate( Places.indexNum2Arr( straw, size) ); //index changed
```

```
      }
  }
}
```

TroopBundle.java

A utility class for gathering information used by the Troop constructor.

```
package SimpleTest;
import java.awt.Color;
class TroopBundle {

  public double xRatio;
  public double yRatio;
  public double radiusRatio;
  public double directionRatio; //radians
  public double speedRatio;
  public Color color;

  TroopBundle(){
      xRatio = Math.random();
      yRatio = Math.random();
      directionRatio = Math.random();
      radiusRatio = Math.random();
      float red = (float)(Math.random() * 0.8 + 0.1);
      float green = (float)(Math.random() * 0.8 + 0.1);
      float blue = (float)(Math.random() * 0.8 + 0.1);
      float alpha = (float)(Math.random() * 0.2 + 0.4);
      color = new Color(red, green, blue, alpha);
      speedRatio = Math.random();
  }
}
```