Supply-Demand Simulation

New Mexico
Supercomputing Challenge
Final Report
1 April 09

Team 96

Members:
Shane Rocha

Mentors:
Jeff Mathis
Ed MacKerrow

Executive Summary

The purpose of this project is to use agent-based programming in Java to simulate the interactions between a small number of producers, sellers, and consumers to create a basic supply-demand chain. All sellers are assumed to be competitive price takers. A seller will raise and lower its price by looking at the sellers that are closest to its grid position. Consumers implement one of three different seller selector strategies. A consumer will either buy from the cheapest seller, the closest seller, or any random seller. A graphical user interface initiated at run-time allows the user to change any of the thirty primary variables used by the program before the simulation starts, allowing the user to see the impact of minor and major changes in market conditions.

Visual output was simplified through the use of the RePast Java library, which provides prefabricated functions that aid in the creation of graphical user interfaces. The Eclipse IDE was implemented to compile and debug the code. The code developed for this project was built upon an economic simulation framework provided by Ed MacKerrow.

The goal will be to eventually increase the complexity and scale in order to produce a more accurate and dynamic model. In this way it is hoped that this project will further our knowledge in the understanding of microeconomics and macroeconomics.

Introduction

The theory of supply and demand is an organizing principle that has been and still is used to explain prices and quantities of goods sold in a market economy. Theoretically, supply is the relation between the price of a good and the quantity available for sale from distributors or suppliers and demand is the quantity that all prospective buyers would be prepared to purchase at each unit of price. The higher the price at which a commodity can be sold, the more of it producers will supply. The higher the price of a product, the less of it people would be willing to purchase.

Current economic models assume that all consumers and sellers are rational and will behave rationally according to a cost-benefit analysis of their economic surroundings and utility desires. Looking at the current economic recession and the causes thereof, we see that the difficulty with these types of models is that they have an inherent computational error. Humans do not always behave rationally, particularly when it comes to spending money.
The question then is thus: how do we more accurately model human behavior? Our answer: with computers of course.

Problem Statement

The purpose of this project is to produce a basic supply-demand simulation without the use of the traditional economic functions and curves through the use of agent-based programming.

Methodology

A supply-demand chain consists of three elements: producers, sellers, and consumers. Producers supply sellers. Sellers act as middlemen. Consumers decide which sellers to purchase from. Everyone is trying to make a profit. While the principles appear simple, the major difficulty in modeling this type of environment is that there is no precedent (that this author knows of) to refer to and compare results to. Instead, all market agent strategies and interfaces were based upon economic principles and modified to represent more closely the world that we see around us.

Consumers

Consumers proved to be the most difficult agents to model. Each consumer is awarded a weekly paycheck, an inventory capacity, and a daily gas consumption rate, each according to a normal distribution based upon whatever variable the user decides to begin with. According to the percentage set by the user, the total number of consumers will be parceled into three different consumer strategy categories.

The random seller selector strategy tells a consumer to choose a seller to purchase from each time by using a random generator to cycle through an array populated by sellers.

Cheapest seller selectors choose a seller by sorting the array of sellers from least to most expensive then choosing the one at the top of the list.

Closest seller selectors compare the sellers' grid coordinates and find the one that is the least distant from their grid location.

Once a consumer has found a seller, it puts itself on RePast's scheduler to buy fill its tank to capacity whenever its gas inventory drops below a certain threshold. When the purchase takes place, the consumer cannot use more than its weekly budget.

Sellers and Producers

Sellers are relatively simple. The main difference between a seller and a consumer is that a seller compares its own price to that of its competitors. In order to undermine its competition, it drops its price just below that of its competitor. It can only drop its price to the point at which it ceases to make money. A producer is a seller that sells to sellers. This is assuming a competitive market.

Results

As is, our miniature economic simulation shows the equivalent of a giant gas war between sellers. As each seller attempts to undercut his neighbors, eventually the price drops so low that the sellers cease making large profits. When a seller attempts to raise the price to make a profit, his competitors briefly raise their prices before undercutting once more. The consumers benefit throughout. Initially, the average agent wealth shows an incredible gap between sellers and consumers. As prices drop, however, the market begins to collapse as sellers can't make a decent profit.

The accuracy of this model is highly speculative as it is largely unfinished.

References

Wessels, Walter J. Economics. New York: Barron's Educational Series, 2000.

Epstein, Joshua M. Generative Social Science: Studies in Agent-Based Computational Modeling. New Jersey: Princeton University Press, 2006.

Michalewicz, Zbigniew, and Dr. David B. Fogel. How to Solve It: Modern Heuristics. New York: Springer-Verlag Berlin Heidelberg, 2000.

Bonabeau, Eric, Marco Dorigo, and Guy Theraulaz. Swarm Intelligence: From Natural to Artificial Systems. New York: Oxford University Press, 1999.

Mathis, Jeff. Personal interview. 16 Feb. 2009.

MacKerrow, Edward. Personal interview. 16 Feb. 2009.

Acknowledgements

I am incredibly grateful for having been blessed with my two incredible mentors, Jeff Mathis and Ed MacKerrow. Even after all other of my other team members had dropped out, Santa Fe High was no longer my sponsor, and the project was on the verge of collapse, their unwavering support and encouragement carried me on.

Source Code

```
/**
 * Copyright 2008, 2009, Edward P. MacKerrow, Agent-Based Computational Economics.  All Rights Reserved.
 */
package com.abcecon.controller;

import java.awt.Color;
import java.awt.Dimension;
import java.net.URL;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;
import java.util.SimpleTimeZone;
import java.util.TimeZone;
import java.util.TreeMap;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.analysis.Histogram;
import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.engine.ScheduleBase;
import uchicago.src.sim.engine.SimModelImpl;
import uchicago.src.sim.gui.DefaultGraphLayout;
import uchicago.src.sim.gui.DisplaySurface;
import uchicago.src.sim.gui.Network2DDisplay;
import uchicago.src.sim.math.Pareto;
import uchicago.src.sim.util.Random;
import cern.jet.random.Normal;
import cern.jet.random.Uniform;

import com.abcecon.model.ConstantInventoryInitializer;
import com.abcecon.model.IProductInitializer;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.NormalDistributionProductDistributor;
import com.abcecon.model.SIMUTIL;
import com.abcecon.model.TIME;
import com.abcecon.model.Transaction;
import com.abcecon.model.buyers.CheapestSellerSelector;
import com.abcecon.model.buyers.ClosestSellerSelector;
import com.abcecon.model.buyers.Consumer;
import com.abcecon.model.buyers.IConsumptionRateAssigner;
import com.abcecon.model.buyers.NormalDistributionConsumptionRateAssigner;
import com.abcecon.model.buyers.RandomSellerSelector;
import com.abcecon.model.buyers.WeeklyWealthUpdater;
import com.abcecon.model.producers.Producer;
```

```
import com.abcecon.model.sellers.Seller;
import com.abcecon.model.sellers.SellerCompetitivePriceTakerStrategy;
import com.abcecon.view.AverageSellerPriceTimeSeriesGraph;

public class GasSupplyDemandSim extends SimModelImpl {
	// this is a Logger, in particular it is one from the Log4J library.  We use it to write out information to help us debug and keep
	//track of things. The old school method was to use System.out.println("HI there, this is a debug statement..."), now we use Loggers since
	//they are more flexible, powerful, and easier to adjust after the simulation is written.
	static Logger LOGGER = Logger.getLogger(GasSupplyDemandSim.class);



	// constants for scheduling events
	public static final double DAYS_TO_UPDATE_WEALTH_HISTOGRAM = 7.0;
	public static final double DAYS_TO_UPDATE_GASBUYER_INVENTORY_HISTOGRAM = 3.0;

	// constant for limiting the max number of Producers so the layouts do not go crazy.
	public static final int MAX_NUMBER_OF_PRODUCERS = 10;

	// trigger levels for distribution of when a buyer agent will decide to buy gas initially.
	public static final double MIN_TRIGGER_LEVEL_TO_BUY = 0.05;  // 5% of the tank ( on fumes )
	public static final double MAX_TRIGGER_LEVEL_TO_BUY = 0.75;  // 75% of the tank

	private static Date startDate;

	/** Here are just saying that each Repast tick of the scheduler corresponds to one second */
	public static final String TIMEPERTICK = "SECOND";
	public static final double TICKS_PER_MINUTE = 60.0;
	public static final double TICKS_PER_HOUR = 60.0 * TICKS_PER_MINUTE;
	public static final double TICKS_PER_DAY = 24.0 * TICKS_PER_HOUR;
	public static final double TICKS_PER_WEEK = 7.0 * TICKS_PER_DAY;
	//after this we get into a mess due to leap years and days per month not being constant.

	public static final double PRODUCER_Y_DISPLAY = 10; // put producer icons at this many pixels from display top
	public static final double SELLER_Y_DISPLAY = 90; // put Seller icons at this many pixels from display top

	// The size of the spatial network display. for some unknown reason when these are less than 500 the spacing of the gasBuyers gets messed up.
	private int displayWidth = 1000;  private int displayHeight = 1000;


	// This name just comes up on the top most frame/window for the RePast Simulation
	private String name = "Gasoline Supply & Demand Simulation";

	/**
	 * The store of ALL transactions, sorted in a TreeMap by the simulation time.
	 */
	private static TreeMap<Double,Transaction> transactions;

	// this is the number of consumers that will be entered via the RePast GUI.
```

```
	private int consumersNumberOf = 12;

	// this is the number of sellersNumberOf that will be entered via the RePast GUI.
	private int sellersNumberOf = 12;

	/**
	 * The total costs per hour of labor for the seller. If they have 2 people working and each makes $8/hr then this would be $16/hr.
	 * This is used to calculate variable costs since the amount of product sold is directly related to how many hours they are open.
	 */
	private double sellerLaborCostsPerHourMean = 2.0 * 8.0;

	private double sellerLaborCostsPerHourStdDev = 2.0;

	private int producersNumberOf = 2;

	/** this is our container of all gasBuyers in the simulation.  It is defined here as an Interface,
	since this makes other classes less dependent, more flexible, to this. */
	private List<MarketAgent> gasBuyers;

	private List<MarketAgent> gasSellers;

	private List<MarketAgent> gasProducers;

	// NOTE **** THE NAMES OF THE FIELDS BELOW ARE INTENTIONALLY "LAME" TO HAVE THEM SHOW UP IN ORDER IN THE REPAST GUI *******

	// ED_TODO change the time dependence to reflect changes in behavior due to price or other.. during the course of the simulation
	private double consumerInitialDailyConsumptionRateMEAN = 1.0; // product units per day, will be fixed once it assigned to each agent
	private double consumerInitialDailyConsumptionRateSTDDEV = 0.1 * consumerInitialDailyConsumptionRateMEAN;

	private double initialMeanProductPrice = 2.00;


	/** Strategy pattern again for assigning different consumption rates to consumers */
	private IConsumptionRateAssigner consumptionRateAssigner;



	// Buyers initial gas inventories
	private double consumerInitialInventoryMEAN = 10.0;
	private double consumerInitialInventorySTDDEV = 2.0;

	// Capacity of the sellers
	private double sellerCapacityMEAN = 1000.0;
	private double sellerCapacitySTDDEV = 20.0;


	// Sellers initial inventories
```

```
        private double sellerInitialInventoryMEAN = 1000.0;
        private double sellerInitalInventorySTDDEV = 20.0;



        // Seller initial price
        private double sellerInitialPriceMEAN = 2.00;
        private double sellerInitialPriceSTDDEV = 0.10;

        // Frequency (in days) for Seller to check whether to update their ask price
        private double sellerTimePeriodDaysToAdjustPriceMEAN = 7.0;
        private double sellerTimePeriodDaysToAdjustPriceSTDDEV = 2.0;

        // Producers initial inventories
        private double producerInitialInventoryMEAN = this.sellersNumberOf * 2.0 * sellerInitialInventoryMEAN;
        private double producerInitialInventorySTDDEV = 0.1 * sellerInitialInventoryMEAN;

        // Producers initial price
        private double producerInitialPriceMEAN = 1.50;
        private double producerInitialPriceSTDDEV = 0.10;

        // Capacity of the consumers
        private double consumerCapacityMEAN = 10.0;
        private double consumerCapacitySTDDEV = 2.0; // gallons

        // these are parameters used to define the Pareto distribution of wealth.  See Wikipedia on "Pareto
Distribution".
        private double consumerWealthMEAN = 50000.00; // assume units of dollars here.
        private double consumerWealthParetoShape = 1.5; // this must be > 1.0


        // the frequency in days that the consumer checks their inventory levels, and buys more if needed.
        private double consumerFrequencyDaysToCheckInventoryMEAN = 7.0;
        private double consumerFrequencyDaysToCheckInventorySTDDEV = 2.0;

        // what fraction of consumers use which strategy
        private double consumerFractionFindCheapestSeller = 0.5;
        private double consumerFractionFindClosestSeller = 0.25;
        private double consumerFractionFindRandomSeller = 0.25;


        // This is the schedule that RePast uses.  It is really like a queue that stores events to happen in the future,
and provides a framework for gasBuyers
        // to schedule future actions
        private Schedule schedule;

        private static Calendar calendar;



        // Displays
        private DisplaySurface spatialDisplaySurface;
        private boolean showNetwork = true;
```

```
	// show random graph network or ordered grid network
	private boolean randomNetworkLayout = false;
	// wealth histogram
	private Histogram wealthHistogram;
	private boolean showWealthHistogram = true;
	// inventory histogram
	private Histogram agentInventoryHistogram;
	private boolean showAgentInventoryHistogram = true;
	Network2DDisplay marketNetworkDisplay;
	// sales price time series
	private boolean showSellerAskPriceTimeSeries = true;
	private AverageSellerPriceTimeSeriesGraph sellerAskPriceSequenceGraph;
	private int updateAvgSellerPriceTimeSeriesGraphFrequencyDays = 14;



	private GasSupplyDemandSim(){
		super();

		// here I am configuring the Logger.  It is really reading a text file that we have set a bunch of
properties
		// for the Logger, for example, what level to print out debug statements etc.
		// The language, or call to do this is obscure, so you really just have to get used to how to use
these loggers...It is not obvious.
		// configure log4j
		URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
		PropertyConfigurator.configure(configURL);

		LOGGER.info("Exiting Simulation() constructor...");

	}


	/* (non-Javadoc)
	 * @see uchicago.src.sim.engine.SimModel#begin()
	 */
	public void begin() {
		LOGGER.info("Entering GasSupplyDemandSim.begin() method...");

		buildModel();
		buildSchedule();
		buildDisplay();

		if(this.showWealthHistogram)
			this.wealthHistogram.display();
		if(this.showAgentInventoryHistogram)
			this.agentInventoryHistogram.display();
	}

	/*
	 * This is a useful RePast method.  You just put the name of the parameters here, that have respective
getters and setters and then RePast
	 * will add these to the startup RePast GUI setup for users to edit.
```

```
	 *
	 * (non-Javadoc)
	 * @see uchicago.src.sim.engine.SimModel#getInitParam()
	 */
	public String[] getInitParam() {
		return new String[] { "consumersNumberOf",
"sellersNumberOf","producersNumberOf","consumerWealthMEAN", "consumerWealthParetoShape",

	"consumerInitialDailyConsumptionRateMEAN","consumerInitialDailyConsumptionRateSTDDEV","initialMe
anProductPrice",
					"consumerInitialInventoryMEAN", "consumerInitialInventorySTDDEV",
"sellerInitialInventoryMEAN", "sellerInitalInventorySTDDEV",
					"producerInitialInventoryMEAN", "producerInitialInventorySTDDEV",

	"consumerInitialDailyConsumptionRateMEAN","ShowWealthHistogram","ShowAgentInventoryHistogram
","ShowNetwork",

	"consumerCapacityMEAN","consumerCapacitySTDDEV","consumerFractionFindClosestSeller","consumerF
ractionFindCheapestSeller","consumerFractionFindRandomSeller",  // ED_2_SHANE I added this variable here to
get the GUI to show it.
					"sellerLaborCostsPerHourMean","sellerLaborCostsPerHourStdDev",
"sellerInitialPriceMEAN", "sellerInitialPriceMEAN",
					"producerInitialPriceMEAN",
"producerInitialPriceSTDDEV","consumerFrequencyDaysToCheckInventoryMEAN","consumerFrequencyDaysToChe
ckInventorySTDDEV",

	"sellerTimePeriodDaysToAdjustPriceMEAN","sellerTimePeriodDaysToAdjustPriceSTDDEV",
"sellerCapacityMEAN", "sellerCapacitySTDDEV","randomNetworkLayout"};
	}

	/* (non-Javadoc)
	 * @see uchicago.src.sim.engine.SimModel#getName()
	 */
	public String getName() {
		return this.name;
	}

	/* (non-Javadoc)
	 * @see uchicago.src.sim.engine.SimModel#getSchedule()
	 */
	public  Schedule getSchedule() {
		return schedule;
	}

	/* Setup is where we "rebuild" the simulation, that is to clear out any existing objects and recreate them.
	 *
	 * (non-Javadoc)
	 * @see uchicago.src.sim.engine.SimModel#setup()
	 */
	public void setup() {

		LOGGER.info("Entering Simulation.setup() method ...");
```

```
		// make sure we wipe out everything and rebuild
		schedule = null;
		this.gasBuyers = null;
		this.gasSellers = null;
		this.gasProducers = null;
		calendar = null;

		this.wealthHistogram = null;
		this.agentInventoryHistogram = null;

		// this is a call to try and force the Java virtual machine to get rid of any objects that it is not using, to free up memory.  It is not guaranteed to listen to me.
		System.gc();

		// Set up a calendar
		setupCalendar();


		// create a new Schedule
		schedule = new Schedule();


		// reinstantiate the containers
		this.gasBuyers = new ArrayList<MarketAgent>();
		this.gasSellers = new ArrayList<MarketAgent>();
		this.gasProducers = new ArrayList<MarketAgent>();
		transactions = new TreeMap<Double, Transaction>();



	}

	/**
	 * This method builds up all the objects required for the simulation to run.
	 */
	public void buildModel(){
		LOGGER.info("Entering Simulation.buildModel() method ...");

		// create new agent instances
		buildGasBuyers();
		buildGasSellers();
		buildGasProducers();

		// assign wealth to the gasBuyers
		initializeWealth();

		// set the gas tank sizes of the gas buyers
		intializeInventoryCapacityOfBuyers();

		// set the initial trigger points, levels of gas left in tank, that the agent will buy gas. this is just the INITIAL trigger points.
		initializeBuyTriggerLevels();
```

```
		// assign initial gas inventories to producers, sellers, and buyers
		this.initializeGasInventories();

		// Assign consumption rates of gasoline to gasBuyers
		this.consumptionRateAssigner = new NormalDistributionConsumptionRateAssigner(this,
this.consumerInitialDailyConsumptionRateMEAN, this.consumerInitialDailyConsumptionRateSTDDEV);
		this.consumptionRateAssigner.assignDailyConsumptionRate(this.gasBuyers);


		// Assign strategies to buyers for choosing sellers
		assignBuyerChooseSellerStrategies();







	}


	/**
	 * Internal method to set the tank sizes (capacities) of all Consumer agents.
	 */
	private void intializeInventoryCapacityOfBuyers(){
		// create a Normal distribution of gas tank sizes for the buyers.
		//ED_TODO check for negative numbers for the mean and std dev here and other places.
		Normal capacityDistribution = new Normal(this.consumerCapacityMEAN,
this.consumerCapacitySTDDEV, Random.getGenerator());
		for(int i = 0; i < gasBuyers.size(); i++){

			MarketAgent buyer = gasBuyers.get(i);
			double draw = -1.0;
			do{
				draw = capacityDistribution.nextDouble();
			}while( draw <= 0.0);
			// then set their capacity
			buyer.setCapacity(draw);
		}

	}

	/**
	 * Internal method to set the tank sizes (capacities) of all Seller agents.
	 */
	private void intializeInventoryCapacityOfSellers(){
		// create a Normal distribution of gas tank sizes for the sellers.
		//ED_TODO check for negative numbers for the mean and std dev here and other places.
		Normal gasTankSizeDistribution = new Normal(this.sellerCapacityMEAN,
this.sellerCapacitySTDDEV, Random.getGenerator());
		for(int i = 0; i < gasSellers.size(); i++){

			MarketAgent seller = gasSellers.get(i);
			double draw = -1.0;
```

```
                        do{
                                draw = gasTankSizeDistribution.nextDouble();
                        }while( draw <= 0.0);
                        // then set their gas tank size
                        seller.setCapacity(draw);
                }

        }


        /**
         * Initialize the inventories of the buyers, sellers, and producers. This is private since we only need this
inside this class.
         */
        private void initializeGasInventories(){

                IProductInitializer consumerInventoryInitializer = new NormalDistributionProductDistributor(this,
this.consumerInitialInventoryMEAN, this.consumerInitialInventorySTDDEV);
                consumerInventoryInitializer.initializeInventories(gasBuyers);


                ConstantInventoryInitializer sellerInventoryInitializer = new ConstantInventoryInitializer(1000.0);
                sellerInventoryInitializer.initializeInventories(gasSellers);

                ConstantInventoryInitializer producerInventoryInitializer = new
ConstantInventoryInitializer(1000.0 * 10.0);
                producerInventoryInitializer.initializeInventories(gasProducers);


        }


        private void initializeBuyTriggerLevels(){
                //
                Uniform triggerDist = new Uniform(GasSupplyDemandSim.MIN_TRIGGER_LEVEL_TO_BUY,
GasSupplyDemandSim.MAX_TRIGGER_LEVEL_TO_BUY,Random.getGenerator());
                for(int i = 0; i < gasBuyers.size(); i++){
                        MarketAgent buyer = gasBuyers.get(i);
                        buyer.setFractionOfCapacityTriggerToBuy(triggerDist.nextDouble());
                }
        }

        /**
         * This builds the display surface for the simulation. I am using RePast stuff here.
         */
        protected void buildDisplay() {

                if( this.showNetwork){
                        this.spatialDisplaySurface = new DisplaySurface(new Dimension(displayWidth,
displayHeight), this, "Market Display");
                        // create the graph layout that holds the gasBuyers that get displayed
                        DefaultGraphLayout layout = new DefaultGraphLayout(displayWidth, displayHeight);
```

```
                layout.getNodeList().addAll(this.gasBuyers);
                layout.getNodeList().addAll(this.gasSellers);
                layout.getNodeList().addAll(this.gasProducers);


                // tell the display surface to display the layout (after wrapping
                // it in a Network2DDisplay
                this.marketNetworkDisplay = new Network2DDisplay(layout);
                spatialDisplaySurface.addDisplayableProbeable(marketNetworkDisplay,"Market
Display");
                this.registerDisplaySurface("display", spatialDisplaySurface);
                spatialDisplaySurface.setBackground(Color.WHITE);
                spatialDisplaySurface.display();

                registerDisplaySurface("market", spatialDisplaySurface);

        }



        // Create a histogram of wealth
        if( this.showWealthHistogram){
                this.wealthHistogram = new Histogram("Wealth", 50, 0.0, 1.0E6, this);
                this.wealthHistogram.createHistogramItem("Wealth", gasBuyers, "getWealth");
                this.wealthHistogram.display();
        }

        // Create a histogram of gas amounts for all gasBuyers
        if( this.showAgentInventoryHistogram){
                this.agentInventoryHistogram = new Histogram("MarketAgent Gas Inventories", 50, 0.0,
40.0, this);
                this.agentInventoryHistogram.createHistogramItem("MarketAgent Gas Inventories",
gasBuyers, "getInventory");
                this.agentInventoryHistogram.display();
        }

        // Create an OpenSequenceGraph of Average Sales Price versus Time
        if( showSellerAskPriceTimeSeries){
                this.sellerAskPriceSequenceGraph = new
AverageSellerPriceTimeSeriesGraph(this,updateAvgSellerPriceTimeSeriesGraphFrequencyDays);

        }


    }


    /**
     * Builds up the schedule. IMPORTANT here NOT to script everything, just try to schedule the initial action
of the agents,
     * then let them schedule their next steps.
     *
     */
```

```
	private void buildSchedule(){
		LOGGER.info("Entering GasSupplyDemandSim.buildSchedule() method...");


		// Each agent has an initialize() method. In these sublcasses of MarketAgent.initialize() method is where agents are supposed to schedule their action

		for(int i = 0; i < gasBuyers.size(); i++){
			MarketAgent buyer = gasBuyers.get(i);
			buyer.initialize();
		}


		for(int i = 0; i < gasSellers.size(); i++){
			MarketAgent seller = gasSellers.get(i);
			seller.initialize();
		}


		for(int i = 0; i < gasProducers.size(); i++){
			MarketAgent producer = gasProducers.get(i);
			producer.initialize();
		}


		// Update the wealth histogram
		if( this.showWealthHistogram){
			// update the wealth histogram display every week
			schedule.scheduleActionAtInterval( TIME.daysToSimulationDuration(DAYS_TO_UPDATE_WEALTH_HISTOGRAM), new BasicAction(){

				@Override
				public void execute() {
					wealthHistogram.step();
				}
			}, ScheduleBase.LAST);
		}


//		// schedule daily gas consumption by all gasBuyers
//		schedule.scheduleActionAtInterval( TIME.daysToSimulationDuration(1.0), new BasicAction(){
//
//			@Override
//			public void execute() {
//				for(int i = 0; i < gasBuyers.size(); i++){
//					MarketAgent gasBuyer = gasBuyers.get(i);
//					double currentGasLevel = gasBuyer.getInventory();
//					if( currentGasLevel >= 0 ){
//						// burn some gas up
//						double amountGasBurnedUpToday = ((Consumer)gasBuyer).getDailyGasConsumptionRate();
//
```

```
//                                                      gasBuyer.setInventory( Math.max(0.0,currentGasLevel -
amountGasBurnedUpToday) );
//                                              }else{
//                                                      //agent.buyGas();
//                                              }
//                                      }
//                              }
//                      });


                        // update the agent gas inventories display every 2 days
                        if( this.showAgentInventoryHistogram){
                                schedule.scheduleActionAtInterval(
TIME.daysToSimulationDuration(DAYS_TO_UPDATE_GASBUYER_INVENTORY_HISTOGRAM), new BasicAction(){

                                        @Override
                                        public void execute() {
                                                agentInventoryHistogram.step();
                                        }
                                }, ScheduleBase.LAST);
                        }



        }



        /**
         * This method is used to create gasBuyers we call this from inside this class only, hence the private.
         */
        private void buildGasBuyers(){



                        if( this.randomNetworkLayout ){
                                //                      // create a uniform distribution for assigning positions randomly to the
gasBuyers (SAVE THIS IN CASE WE GO BACK TO RANDOM GRID)
                                        Uniform uniDistX = new Uniform(10.0, this.displayWidth - 10.0,
Random.getGenerator());
                                        Uniform uniDistY = new Uniform(10.0, this.displayHeight - 10.0,
Random.getGenerator());

                                        for( int i = 0; i < consumersNumberOf; i++){
                                                double xPos = uniDistX.nextDouble();
                                                double yPos = uniDistY.nextDouble();
                                                        MarketAgent gasBuyer = new Consumer(this, xPos,
yPos  );
                                                        // store the agent in our container
                                                        this.gasBuyers.add( gasBuyer);
                                        }
                        }else{
```

```
                        // Spatial layout

                        // try to line up the buyers in columns under the sellers
                        double xSpacing = this.displayWidth / this.sellersNumberOf;
                        int numberOfRows = this.consumersNumberOf / sellersNumberOf;
                        double ySpacing = (this.displayHeight - GasSupplyDemandSim.SELLER_Y_DISPLAY - 50.0)
/ numberOfRows;

                        for( int i = 0; i < consumersNumberOf; i++){
                                // create a new agent with randomly drawn coordinates.
                                for( int j = 0; j < numberOfRows; j++){
                                        MarketAgent gasBuyer = new Consumer(this, i * xSpacing +
xSpacing/4, 50.0 + SELLER_Y_DISPLAY + j * ySpacing + ySpacing / 2  );

                                        // store the agent in our container
                                        this.gasBuyers.add( gasBuyer);
                                }
                        }
                }


                Normal checkGasFrequencyDaysDist = new
Normal(this.consumerFrequencyDaysToCheckInventorySTDDEV,
this.consumerFrequencyDaysToCheckInventorySTDDEV, Random.getGenerator());

                // set the wealth updater implementation for all the buyers.
                for(int i = 0; i < this.gasBuyers.size(); i++){
                        MarketAgent buyer = this.gasBuyers.get(i);
                        buyer.setWealthUpdater(new WeeklyWealthUpdater());
                        // set check gas freq
                        buyer.setFrequencyDaysCheckInventory( Math.max(
checkGasFrequencyDaysDist.nextDouble(), 1.0 ) );
                }


                LOGGER.info("Exiting GasSupplyDemandSim.buildGasBuyers() with "+ consumersNumberOf + "
created ");


        }

        /**
         * Assigns strategy to BUYERS of which Seller they will buy gas from.
         *
         */
        private void assignBuyerChooseSellerStrategies() {
                // setup the ISupplierSelector strategies on the buyers
                int numOfFindCheapestSellers = (int) Math.round( this.consumersNumberOf *
this.consumerFractionFindCheapestSeller );
                int numOfFindClosestSellers = (int) Math.round( this.consumersNumberOf *
this.consumerFractionFindClosestSeller );
```

```
		int numOfFindRandomSellers = this.consumersNumberOf - ( numOfFindCheapestSellers +
numOfFindClosestSellers);

		ArrayList<MarketAgent> listOfBuyers = new ArrayList<MarketAgent>(this.gasBuyers);

		ArrayList<MarketAgent> listOfBuyersWhoNeedSelectSellerStrategies = new
ArrayList<MarketAgent>(this.gasBuyers); // use this to keep track of who still needs a strategy


		// Set the number of CheapestSellerBuyers
		for( int i = 0; i < numOfFindCheapestSellers; i++){
			MarketAgent findCheapestSellerBuyer = listOfBuyers.get(i);
			// set the implementor
			(findCheapestSellerBuyer).setSellerSelector(new CheapestSellerSelector(this) );

			// then remove them for our list of still need strategy
			listOfBuyersWhoNeedSelectSellerStrategies.remove(findCheapestSellerBuyer);

		}

		// Set the number of ClosestSellerBuyers
		for( int i = 0; i < listOfBuyersWhoNeedSelectSellerStrategies.size(); i++){
			MarketAgent findClosestSellerBuyer =
listOfBuyersWhoNeedSelectSellerStrategies.get(i);

			// ED_2_SHANE start
			// then remove them for our list of still need strategy
			(findClosestSellerBuyer).setSellerSelector(new
ClosestSellerSelector(this,findClosestSellerBuyer) );

			listOfBuyersWhoNeedSelectSellerStrategies.remove(findClosestSellerBuyer);
			// ED_2_SHANE end

		}

		// ED_2_SHANE start
		// The rest of the list is the other strategy
		for(int i = 0; i < listOfBuyersWhoNeedSelectSellerStrategies.size(); i++){
			com.abcecon.model.MarketAgent findRandomSellerBuyer =
listOfBuyersWhoNeedSelectSellerStrategies.get(i);
			// set the implementor
			findRandomSellerBuyer.setSellerSelector(new RandomSellerSelector(this,
findRandomSellerBuyer));


		}

		// we are done with this list now
		listOfBuyers.clear();
		listOfBuyers = null;

		listOfBuyersWhoNeedSelectSellerStrategies.clear();
		listOfBuyersWhoNeedSelectSellerStrategies = null;
```

```
		// ED_2_SHANE end

	}



	/**
	 * This method is used to create gasSellers we call this from inside this class only, hence the private.
	 */
	private void buildGasSellers(){


		double xSpacing = this.displayWidth / this.sellersNumberOf;

		if( this.randomNetworkLayout ){
			//			// create a uniform distribution for assigning positions randomly to the gasBuyers (SAVE THIS IN CASE WE GO BACK TO RANDOM GRID)
			Uniform uniDistX = new Uniform(2.0, this.displayWidth - 2.0, Random.getGenerator());
			Uniform uniDistY = new Uniform(2.0, this.displayHeight - 2.0, Random.getGenerator());

			for( int i = 0; i < sellersNumberOf; i++){
				double xPos = uniDistX.nextDouble();
				double yPos = uniDistY.nextDouble();
					MarketAgent gasSeller = new Seller(this, xPos, yPos );

					// store the agent in our container
					this.gasSellers.add( gasSeller);
			}
		}else{
			//			// create a uniform distribution for assigning positions randomly to the gasSellers
			Uniform uniDistX = new Uniform( 10.0, this.displayWidth - 10.0, Random.getGenerator() );
			Uniform uniDistY = new Uniform( 10.0, this.displayHeight - 10.0, Random.getGenerator() );

			// Setup spatial coordinates.
			for( int i= 0; i < sellersNumberOf; i++){
				// create a new agent with randomly drawn coordinates.
				MarketAgent gasSeller = new Seller(this, i * xSpacing + xSpacing/2 , GasSupplyDemandSim.SELLER_Y_DISPLAY);

				// store the agent in our container
				this.gasSellers.add(gasSeller);
			}

		}
```

```
		//initialize  prices
		Normal priceDist = new
Normal(this.sellerInitialPriceMEAN,this.sellerInitialPriceSTDDEV,Random.getGenerator());

		// initialize the time period that each seller rechecks its askPrice and possibly changes it
		Normal freqencyAdjustAskPriceDistribution = new Normal(
this.sellerTimePeriodDaysToAdjustPriceMEAN, this.sellerTimePeriodDaysToAdjustPriceSTDDEV,
Random.getGenerator());

		for(int i = 0; i < this.gasSellers.size(); i++){
			MarketAgent seller = this.gasSellers.get(i);

			// set initial ask price
			seller.setAskPrice(priceDist.nextDouble());

			// assign the ask price strategies (we only use one strategy for now)
			seller.setAskPriceStrategy(new SellerCompetitivePriceTakerStrategy(seller));

			// set the frequency in days that each seller will adjust their ask price.
			double daysToCheckAskPrice = Math.max(1.0,
freqencyAdjustAskPriceDistribution.nextDouble());
			((Seller)seller).setTimePeriodDaysToAdjustAskPrice(daysToCheckAskPrice);

		}



		LOGGER.info("Exiting GasSupplyDemandSim.buildGasSellers() with "+ sellersNumberOf + "
created ");


	}


	/**
	 * This method is used to create gasProducers we call this from inside this class only, hence the private.
	 */
	private void buildGasProducers(){

		//			// create a uniform distribution for assigning positions randomly to the
gasSellers
		//			Uniform uniDistX = new Uniform(0, this.displayWidth - 1,
Random.getGenerator());
		//			Uniform uniDistY = new Uniform(0, this.displayHeight - 1,
Random.getGenerator());

		double xSpacing = this.displayWidth / this.producersNumberOf;


		for( int i= 0; i < producersNumberOf; i++){
			// create a new agent with randomly drawn coordinates.
```

```
                        Producer gasProducer = new Producer(this, i * xSpacing + xSpacing/2 ,
GasSupplyDemandSim.PRODUCER_Y_DISPLAY);

                        // store the agent in our container
                        this.gasProducers.add( gasProducer);
                }

                //initialize gas prices
                Normal priceDist = new
Normal(this.producerInitialPriceMEAN,this.producerInitialPriceSTDDEV,Random.getGenerator());
                for(int i = 0; i < this.gasProducers.size(); i++){
                        MarketAgent producer = this.gasProducers.get(i);
                        ((Producer) producer).setAskPrice(priceDist.nextDouble());
                }

                LOGGER.info("Exiting GasSupplyDemandSim.buildGasProducers() with "+ producersNumberOf +
" created ");


        }



        /**
         * Assign wealth to each agent based on a Pareto Distribution (most of the wealth is owned by a minority
         *  of the population -- sad but true ).
         *
         */
        private void initializeWealth(){

                //set up the Pareto distribution based on the input shape parameter and the mean.  See
Wikipedia for what a Pareto distribution is.
                double paretoScale = ( (consumerWealthParetoShape - 1.0 ) / consumerWealthParetoShape ) *
consumerWealthMEAN;

                // here we use the RePast Pareto distribution, if this does not work check out the Cern Power
Law distribution
                Pareto wealthDist = new Pareto(paretoScale, consumerWealthParetoShape,
Random.getGenerator());

                // change to format of dollars
                NumberFormat dollarsFormat = NumberFormat.getCurrencyInstance();

                // now go through the gasBuyers and assign random wealth to them.
                for(int i = 0; i < gasBuyers.size(); i++){
                        MarketAgent gasBuyer = gasBuyers.get(i);
                        double wealth = wealthDist.nextDouble();
                        gasBuyer.setWealth(wealth);

                        LOGGER.debug("MarketAgent: " +gasBuyer.getId()+" has new wealth of
"+dollarsFormat.format(wealth));
                }
        }
```

```
	/**
	 * Sets up a Calendar to help us have meaningful dates in the simulation. The alternative to this is that
	 * our time steps and "dates" would simply be clock ticks of the computer.  Here we utilize some of the
functionality
	 * that the Java libraries provide us for free.
	 *
	 */
	private void setupCalendar(){
		// get the supported ids for GMT-07:00 (Mountain Standard Time)
		String[] ids = TimeZone.getAvailableIDs(-7 * 60 * 60 * 1000);
		// if no ids were returned, something is wrong. get out.
		if (ids.length == 0)
			System.exit(0);

		// begin output
		System.out.println("Current Time");

		// create a Mountain Standard Time time zone
		SimpleTimeZone mdt = new SimpleTimeZone(-8 * 60 * 60 * 1000, ids[0]);

		// set up rules for daylight savings time
		mdt.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);
		mdt.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);

		// create a GregorianCalendar with the Mountain Daylight time zone
		// and the current date and time
		calendar = new GregorianCalendar(mdt);
		GasSupplyDemandSim.startDate = new Date();
		calendar.setTime(GasSupplyDemandSim.startDate);


		// a debug statement to print out when we run.
		LOGGER.debug("Calendar time at setup is: " + calendar.getTime().toString());

	}



	/**
	 * @return the calendar
	 */
	public static Calendar getCalendar() {
		return calendar;
	}


	/**
	 * Returns the current time of the simulation Calendar. Assumes that the fundamental time step of the
simulation is in milliseconds.
```

```
	 * This should only be used to get a calendar date, since if you call
<code>getCurrentDate().getTime()</code> you will get the number
	 * of millisecs from 1970.
	 *
	 * Important, this method sets the {@link GasSupplyDemandSim#calendar} time to the current simulation
time, offset from the start of the simulation.
	 * @return
	 */
	public Date getCurrentDate(){
		// get the current clock tick and make sure to add it to the starting date
		long currentSimTime = GasSupplyDemandSim.startDate.getTime() + (new
Double(schedule.getCurrentTime()).longValue() );

		// set the calendar (the calendar is not the scheduler, it is not dynamically/automatically
updated)
		calendar.setTime(new Date(currentSimTime));

		return calendar.getTime();
	}



	/**
	 * @return the consumerFractionFindCheapestSeller
	 */
	public double getConsumerFractionFindCheapestSeller() {
		return consumerFractionFindCheapestSeller;
	}

	/**
	 * @param consumerFractionFindCheapestSeller the consumerFractionFindCheapestSeller to set
	 */
	public void setConsumerFractionFindCheapestSeller(double _frac) {
		_frac = Math.min(_frac, 1.0); // force to be < 1.0
		_frac = Math.max(_frac, 0.0); // force to be > 0.0

		this.consumerFractionFindCheapestSeller = _frac;
		// force the other fraction to add to one
		this.consumerFractionFindClosestSeller = 1.0 - consumerFractionFindCheapestSeller;
		this.consumerFractionFindRandomSeller = 1.0 - consumerFractionFindCheapestSeller;
	}

	/**
	 * @return the consumerFractionFindClosestSeller
	 */
	public double getConsumerFractionFindClosestSeller() {
		return consumerFractionFindClosestSeller;
	}

	/**
	 * @param consumerFractionFindClosestSeller the consumerFractionFindClosestSeller to set
	 */
```

```
	public void setConsumerFractionFindClosestSeller(double _frac) {
		this.consumerFractionFindClosestSeller = _frac;
		_frac = Math.min(_frac, 1.0); // force to be < 1.0
		_frac = Math.max(_frac, 0.0); // force to be > 0.0

		this.consumerFractionFindClosestSeller = _frac;
		// force the other fraction to add to one
		this.consumerFractionFindCheapestSeller = 1.0 - consumerFractionFindClosestSeller;
		this.consumerFractionFindRandomSeller = 1.0 - consumerFractionFindClosestSeller;

	}

	/**
	 * @return the consumerFractionFindRandomSeller
	 */
	public double getConsumerFractionFindRandomSeller() {
		return consumerFractionFindRandomSeller;
	}

	/**
	 * @param consumerFractionFindRandomSeller the consumerFractionFindClosestSeller to set
	 */
	public void setConsumerFractionFindRandomSeller(double _frac) {
		this.consumerFractionFindRandomSeller = _frac;
		_frac = Math.min(_frac, 1.0); // force to be < 1.0
		_frac = Math.max(_frac, 0.0); // force to be > 0.0

		this.consumerFractionFindRandomSeller = _frac;
		// force the other fraction to add to one
		this.consumerFractionFindCheapestSeller = 1.0 - consumerFractionFindRandomSeller;
		this.consumerFractionFindClosestSeller = 1.0 - consumerFractionFindRandomSeller;

	}

	/**
	 * @return the consumerInitialDailyConsumptionRateMEAN
	 */
	public double getConsumerInitialDailyConsumptionRateMEAN() {
		return consumerInitialDailyConsumptionRateMEAN;
	}

	/**
	 * @param consumerInitialDailyConsumptionRateMEAN the consumerInitialDailyConsumptionRateMEAN
to set
	 */
	public void setConsumerInitialDailyConsumptionRateMEAN(double meanWeeklyGasConsumption) {
		this.consumerInitialDailyConsumptionRateMEAN = meanWeeklyGasConsumption;
	}

	/**
	 * @return the initialMeanProductPrice
	 */
	public double getInitialMeanProductPrice() {
```

```
		return initialMeanProductPrice;
}

/**
 * @param initialMeanProductPrice the initialMeanProductPrice to set
 */
public void setInitialMeanProductPrice(double meanInitialGasPrice) {
		this.initialMeanProductPrice = meanInitialGasPrice;
}

/**
 * @return the consumersNumberOf
 */
public int getConsumersNumberOf() {
		return consumersNumberOf;
}

/**
 * @param consumersNumberOf the consumersNumberOf to set
 */
public void setConsumersNumberOf(int numberOfAgents) {
		this.consumersNumberOf = numberOfAgents;
}



/**
 * @return the consumerWealthMEAN
 */
public double getConsumerWealthMEAN() {
		return consumerWealthMEAN;
}

/**
 * @param consumerWealthMEAN the consumerWealthMEAN to set
 */
public void setConsumerWealthMEAN(double meanWealth) {
		this.consumerWealthMEAN = meanWealth;
}

/**
 * @return the consumerWealthParetoShape
 */
public double getConsumerWealthParetoShape() {
		return consumerWealthParetoShape;
}

/**
 * @param consumerWealthParetoShape the consumerWealthParetoShape to set
 */
public void setConsumerWealthParetoShape(double paretoWealthShape) {
		this.consumerWealthParetoShape = paretoWealthShape;
}
```

```
	/**
	 * @return the consumerFrequencyDaysToCheckInventorySTDDEV
	 */
	public double getConsumerFrequencyDaysToCheckInventorySTDDEV() {
		return consumerFrequencyDaysToCheckInventoryMEAN;
	}


	/**
	 * @param consumerFrequencyDaysToCheckInventorySTDDEV the
consumerFrequencyDaysToCheckInventorySTDDEV to set
	 */
	public void setConsumerFrequencyDaysToCheckInventorySTDDEV(
				double buyerFrequencyDaysToCheckGasTankLevelMEAN) {
		this.consumerFrequencyDaysToCheckInventorySTDDEV =
buyerFrequencyDaysToCheckGasTankLevelMEAN;
	}


	/**
	 * @return the buyerFrequencyDaysToCheckGasTankLevelSTDDEV
	 */
	public double getBuyerFrequencyDaysToCheckGasTankLevelSTDDEV() {
		return consumerFrequencyDaysToCheckInventoryMEAN;
	}


	/**
	 * @param buyerFrequencyDaysToCheckGasTankLevelSTDDEV the
buyerFrequencyDaysToCheckGasTankLevelSTDDEV to set
	 */
	public void setBuyerFrequencyDaysToCheckGasTankLevelSTDDEV(
				double buyerFrequencyDaysToCheckGasTankLevelSTDDEV) {
		this.consumerFrequencyDaysToCheckInventorySTDDEV =
buyerFrequencyDaysToCheckGasTankLevelSTDDEV;
	}


//	/**
//	 * @return the calendar
//	 */
//	public static Calendar getCalendar() {
//		return calendar;
//	}


	/**
	 * @return the wealthHistogram
	 */
```

```
public Histogram getWealthHistogram() {
	return wealthHistogram;
}



/**
 * @return the sellersNumberOf
 */
public int getSellersNumberOf() {
	return sellersNumberOf;
}



/**
 * @return the consumptionRateAssigner
 */
public IConsumptionRateAssigner getConsumptionRateAssigner() {
	return consumptionRateAssigner;
}

/**
 * @param consumptionRateAssigner the consumptionRateAssigner to set
 */
public void setConsumptionRateAssigner(
			IConsumptionRateAssigner gasConsumptionRateAssigner) {
	this.consumptionRateAssigner = gasConsumptionRateAssigner;
}




public double getConsumerInitialDailyConsumptionRateSTDDEV() {
	return consumerInitialDailyConsumptionRateSTDDEV;
}

public void setConsumerInitialDailyConsumptionRateSTDDEV(
			double stdDevDailyGasConsumptionRate) {
	this.consumerInitialDailyConsumptionRateSTDDEV = stdDevDailyGasConsumptionRate;
}


/**
 * @return the consumerInitialInventoryMEAN
 */
public double getConsumerInitialInventoryMEAN() {
	return consumerInitialInventoryMEAN;
}

/**
 * @param consumerInitialInventoryMEAN the consumerInitialInventoryMEAN to set
 */
```

```
public void setConsumerInitialInventoryMEAN(double meanInitialAgentGas) {
	this.consumerInitialInventoryMEAN = meanInitialAgentGas;
}

/**
 * @return the consumerInitialInventorySTDDEV
 */
public double getConsumerInitialInventorySTDDEV() {
	return consumerInitialInventorySTDDEV;
}

/**
 * @param consumerInitialInventorySTDDEV the consumerInitialInventorySTDDEV to set
 */
public void setConsumerInitialInventorySTDDEV(double stdDevInitialAgentGas) {
	this.consumerInitialInventorySTDDEV = stdDevInitialAgentGas;
}

/**
 * @return the sellerInitialInventoryMEAN
 */
public double getSellerInitialInventoryMEAN() {
	return sellerInitialInventoryMEAN;
}

/**
 * @param sellerInitialInventoryMEAN the sellerInitialInventoryMEAN to set
 */
public void setSellerInitialInventoryMEAN(double meanInitialSellerGas) {
	this.sellerInitialInventoryMEAN = meanInitialSellerGas;
}

/**
 * @return the sellerInitalInventorySTDDEV
 */
public double getSellerInitalInventorySTDDEV() {
	return this.sellerInitalInventorySTDDEV;
}

/**
 * @param sellerInitalInventorySTDDEV the stdInitialSellerGasInventory to set
 */
public void setSellerInitalInventorySTDDEV(double stdDevInitialSellerGas) {
	this.sellerInitalInventorySTDDEV = stdDevInitialSellerGas;
}


/**
 * @return the sellerInitialPriceMEAN
 */
public double getSellerInitialPriceMEAN() {
	return sellerInitialPriceMEAN;
}
```

```
/**
 * @param sellerInitialPriceMEAN the sellerInitialPriceMEAN to set
 */
public void setSellerInitialPriceMEAN(double sellerInitialGasPriceMean) {
	this.sellerInitialPriceMEAN = sellerInitialGasPriceMean;
}

/**
 * @return the sellerInitialPriceSTDDEV
 */
public double getSellerInitialPriceSTDDEV() {
	return sellerInitialPriceSTDDEV;
}

/**
 * @param sellerInitialPriceSTDDEV the sellerInitialPriceSTDDEV to set
 */
public void setSellerInitialPriceSTDDEV(double sellerInitialGasPriceStdDev) {
	this.sellerInitialPriceSTDDEV = sellerInitialGasPriceStdDev;
}




/**
 * Whether or not to show the market network as ordered on a grid, or randomly laid out.
 *
 * @return the randomNetworkLayout
 */
public boolean isRandomNetworkLayout() {
	return randomNetworkLayout;
}


/**
 * Whether or not to show the market network as ordered on a grid, or randomly laid out.
 *
 * @param randomNetworkLayout the randomNetworkLayout to set
 */
public void setRandomNetworkLayout(boolean randomNetworkLayout) {
	this.randomNetworkLayout = randomNetworkLayout;
}


/**
 * @return the sellerCapacityMEAN
 */
public double getSellerCapacityMEAN() {
	return sellerCapacityMEAN;
}
```

```
/**
 * @param sellerCapacityMEAN the sellerCapacityMEAN to set
 */
public void setSellerCapacityMEAN(double sellerCapacityMEAN) {
	this.sellerCapacityMEAN = sellerCapacityMEAN;
}


/**
 * @return the sellerCapacitySTDDEV
 */
public double getSellerCapacitySTDDEV() {
	return sellerCapacitySTDDEV;
}


/**
 * @param sellerCapacitySTDDEV the sellerCapacitySTDDEV to set
 */
public void setSellerCapacitySTDDEV(double sellerCapacitySTDDEV) {
	this.sellerCapacitySTDDEV = sellerCapacitySTDDEV;
}


/**
 * @return the consumerCapacityMEAN
 */
public double getConsumerCapacityMEAN() {
	return consumerCapacityMEAN;
}

/**
 * @param consumerCapacityMEAN the consumerCapacityMEAN to set
 */
public void setConsumerCapacityMEAN(double meanGasBuyerTankSize) {
	this.consumerCapacityMEAN = meanGasBuyerTankSize;
}

/**
 * @return the consumerCapacitySTDDEV
 */
public double getConsumerCapacitySTDDEV() {
	return consumerCapacitySTDDEV;
}

/**
 * @param consumerCapacitySTDDEV the consumerCapacitySTDDEV to set
 */
public void setConsumerCapacitySTDDEV(double stdDevGasBuyerTankSize) {
	this.consumerCapacitySTDDEV = stdDevGasBuyerTankSize;
}

/**
```

```
 * @return the producerInitialInventoryMEAN
 */
public double getProducerInitialInventoryMEAN() {
	return producerInitialInventoryMEAN;
}

/**
 * @param producerInitialInventoryMEAN the producerInitialInventoryMEAN to set
 */
public void setProducerInitialInventoryMEAN(
			double meanInitialProducerGasInventory) {
	this.producerInitialInventoryMEAN = meanInitialProducerGasInventory;
}

/**
 * @return the producerInitialInventorySTDDEV
 */
public double getProducerInitialInventorySTDDEV() {
	return producerInitialInventorySTDDEV;
}

/**
 * @param producerInitialInventorySTDDEV the producerInitialInventorySTDDEV to set
 */
public void setProducerInitialInventorySTDDEV(
			double stdInitialProducerGasInventory) {
	this.producerInitialInventorySTDDEV = stdInitialProducerGasInventory;
}

/**
 * @param sellersNumberOf the sellersNumberOf to set
 */
public void setSellersNumberOf(int numberOfGasSellers) {
	this.sellersNumberOf = numberOfGasSellers;
}



/**
 * @return the sellerLaborCostsPerHourMean
 */
public double getSellerLaborCostsPerHourMean() {
	return sellerLaborCostsPerHourMean;
}

/**
 * @param sellerLaborCostsPerHourMean the sellerLaborCostsPerHourMean to set
 */
public void setSellerLaborCostsPerHourMean(double sellerLaborCostsPerHourMean) {
	this.sellerLaborCostsPerHourMean = sellerLaborCostsPerHourMean;
}

/**
```

```
	 * @return the sellerLaborCostsPerHourStdDev
	 */
	public double getSellerLaborCostsPerHourStdDev() {
		return sellerLaborCostsPerHourStdDev;
	}

	/**
	 * @param sellerLaborCostsPerHourStdDev the sellerLaborCostsPerHourStdDev to set
	 */
	public void setSellerLaborCostsPerHourStdDev(
			double sellerLaborCostsPerHourStdDev) {
		this.sellerLaborCostsPerHourStdDev = sellerLaborCostsPerHourStdDev;
	}



	/**
	 * The mean number of days for sellers to check and possibly change their current ask price.
	 *
	 * @return the sellerTimePeriodDaysToAdjustPriceMEAN
	 */
	public double getSellerTimePeriodDaysToAdjustPriceMEAN() {
		return sellerTimePeriodDaysToAdjustPriceMEAN;
	}


	/**
	 * The mean number of days for sellers to check and possibly change their current ask price
	 *
	 * @param sellerTimePeriodDaysToAdjustPriceMEAN the sellerTimePeriodDaysToAdjustPriceMEAN to set
	 */
	public void setSellerTimePeriodDaysToAdjustPriceMEAN(
			double sellerTimePeriodDaysToAdjustPriceMEAN) {
		this.sellerTimePeriodDaysToAdjustPriceMEAN = sellerTimePeriodDaysToAdjustPriceMEAN;
	}


	/**
	 * @return the sellerTimePeriodDaysToAdjustPriceSTDDEV
	 */
	public double getSellerTimePeriodDaysToAdjustPriceSTDDEV() {
		return sellerTimePeriodDaysToAdjustPriceSTDDEV;
	}


	/**
	 * @param sellerTimePeriodDaysToAdjustPriceSTDDEV the sellerTimePeriodDaysToAdjustPriceSTDDEV to
set
	 */
	public void setSellerTimePeriodDaysToAdjustPriceSTDDEV(
			double sellerTimePeriodDaysToAdjustPriceSTDDEV) {
		this.sellerTimePeriodDaysToAdjustPriceSTDDEV = sellerTimePeriodDaysToAdjustPriceSTDDEV;
	}
```

```
/**
 * @return the displayWidth
 */
public int getDisplayWidth() {
	return displayWidth;
}

/**
 * @param displayWidth the displayWidth to set
 */
public void setDisplayWidth(int displayWidth) {
	this.displayWidth = displayWidth;
}

/**
 * @return the displayHeight
 */
public int getDisplayHeight() {
	return displayHeight;
}

/**
 * @param displayHeight the displayHeight to set
 */
public void setDisplayHeight(int displayHeight) {
	this.displayHeight = displayHeight;
}

/**
 * @return the gasBuyers
 */
public List<MarketAgent> getGasBuyers() {
	return gasBuyers;
}




/**
 * @return the producersNumberOf
 */
public int getProducersNumberOf() {
	return producersNumberOf;
}

/**
 * Sets the number of Producers in the Simulation.  We limit the max number of Producers.
 *
 * @param producersNumberOf the producersNumberOf to set
 */
public void setProducersNumberOf(int numberOfGasProducers) {
```

```
		int num = Math.min(numberOfGasProducers, MAX_NUMBER_OF_PRODUCERS);
		this.producersNumberOf = Math.max(0,num);
}



/**
 * @return the producerInitialPriceMEAN
 */
public double getProducerInitialPriceMEAN() {
		return producerInitialPriceMEAN;
}


/**
 * @param producerInitialPriceMEAN the producerInitialPriceMEAN to set
 */
public void setProducerInitialPriceMEAN(double producerInitialGasPriceMEAN) {
		this.producerInitialPriceMEAN = producerInitialGasPriceMEAN;
}


/**
 * @return the producerInitialPriceSTDDEV
 */
public double getProducerInitialPriceSTDDEV() {
		return producerInitialPriceSTDDEV;
}


/**
 * @param producerInitialPriceSTDDEV the producerInitialPriceSTDDEV to set
 */
public void setProducerInitialPriceSTDDEV(
				double producerInitialGasPriceSTDDEV) {
		this.producerInitialPriceSTDDEV = producerInitialGasPriceSTDDEV;
}


/**
 * @return the gasSellers
 */
public List<MarketAgent> getGasSellers() {
		return gasSellers;
}

/**
 * @return the gasProducers
 */
public List<MarketAgent> getGasProducers() {
		return gasProducers;
}
```

```
public DisplaySurface getSpatialDisplaySurface() {
	return spatialDisplaySurface;
}

public void setSpatialDisplaySurface(DisplaySurface spatialDisplaySurface) {
	this.spatialDisplaySurface = spatialDisplaySurface;
}



/**
 * @return the startDate
 */
public static Date getStartDate() {
	return startDate;
}


/**
 * @return the transactions
 */
public static TreeMap<Double, Transaction> getTransactions() {
	return transactions;
}

public boolean getShowNetwork() {
	return showNetwork;
}

public void setShowNetwork(boolean showNetwork) {
	this.showNetwork = showNetwork;
}

public boolean getShowWealthHistogram() {
	return showWealthHistogram;
}

public void setShowWealthHistogram(boolean showWealthHistogram) {
	this.showWealthHistogram = showWealthHistogram;
}

public boolean getShowAgentInventoryHistogram() {
	return showAgentInventoryHistogram;
}

public void setShowAgentInventoryHistogram(boolean showAgentGasHistogram) {
	this.showAgentInventoryHistogram = showAgentGasHistogram;
}

/**
 * @param name the name to set
 */
```

```
	public void setName(String name) {
		this.name = name;
	}



	/**
	 * The main method to run the simulation.
	 *
	 * @param args the command line arguments if there are any.
	 */
	public static void main(String[] args) {
		uchicago.src.sim.engine.SimInit init = new uchicago.src.sim.engine.SimInit();
		GasSupplyDemandSim sim = createGasSupplyDemandSim();
		init.loadModel(sim, "", false);
	}


	public static GasSupplyDemandSim createGasSupplyDemandSim() {
		return new GasSupplyDemandSim();
	}




}

package com.abcecon.model;

import java.util.Comparator;

public class AskPriceComparator implements Comparator<Transaction>{

	/* (non-Javadoc)
	 * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
	 */
	public int compare(Transaction obj1, Transaction obj2) {
		if( (obj1 instanceof Transaction) && (obj2 instanceof Transaction) ){
			double askPrice1 = (obj1).getAskPrice(); // cast to get the ask price, Probably don't need
this with Java Generics.
			double askPrice2 = (obj2).getAskPrice(); // cast to get the ask price
			if( askPrice1 > askPrice2){
				return 1;
			}else if( askPrice1 < askPrice2){
				return -1;
			}else{
				return 0;
			}


		}else{
			// should not get here
			return 0;
```

```
                }

        }
}

package com.abcecon.model;

import java.util.Comparator;

class BidPriceComparator implements Comparator<Transaction>{



        /* (non-Javadoc)
         * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
         */
        public int compare(Transaction obj1, Transaction obj2) {
                if( (obj1 instanceof Transaction) && (obj2 instanceof Transaction) ){
                        double bidPrice1 = (obj1).getBidPrice(); // cast to get the bid price, Probably don't need
this with Java Generics.
                        double bidPrice2 = (obj2).getBidPrice(); // cast to get the bid price
                        if( bidPrice1 > bidPrice2){
                                return 1;
                        }else if( bidPrice1 < bidPrice2){
                                return -1;
                        }else{
                                return 0;
                        }


                }else{
                        return 0;
                }

        }
}

package com.abcecon.model;

import java.net.URL;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.SimModel;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.buyers.BuyProductAction;

public class CheckInventoryAction extends BasicAction{
```

```
	Logger LOGGER = Logger.getLogger(CheckInventoryAction.class);


	MarketAgent buyer;

	public CheckInventoryAction(MarketAgent _buyer){
		super();
		// Set up the Log4j logger
		URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
		PropertyConfigurator.configure(configURL);

		// turn LOGGER off for a while
		LOGGER.setLevel(Level.OFF);


		this.buyer = _buyer;
	}

	/* (non-Javadoc)
	 * @see uchicago.src.sim.engine.BasicAction#execute()
	 */
	@Override
	public void execute() {

		SimModel sim = buyer.getSim();

		double currentSimulationTime  = sim.getSchedule().getCurrentTime();

		// check how much inventory we have relative to our trigger level to buy
		if( buyer.getInventory() < ( buyer.getCapacity() * buyer.getFractionOfCapacityTriggerToBuy() )){

			// then I need to buy product to replenish my inventory. Schedule it
			if( LOGGER.isDebugEnabled()){
				GasSupplyDemandSim.getCalendar().setTimeInMillis((new
Double(currentSimulationTime + MarketAgent.DELTA_TIME_STEP)).longValue());
				LOGGER.debug( ((GasSupplyDemandSim)sim).getCurrentDate() +": " +
buyer.getId() +
							" is scheduling a buy inventory action on " +
GasSupplyDemandSim.getCalendar().getTime().toString()  );
			}
			// schedule future action to buy inventory NOW
			sim.getSchedule().scheduleActionAt( currentSimulationTime +
MarketAgent.DELTA_TIME_STEP, new BuyProductAction(buyer) );

		}else{
			// then have them check there gas in a few days
			sim.getSchedule().scheduleActionAt(currentSimulationTime +
TIME.daysToSimulationDuration( buyer.getFrequencyDaysCheckInventory() ), new CheckInventoryAction(buyer));

		}

	}
```

```
}

package com.abcecon.model;

import java.util.List;

public class ConstantInventoryInitializer implements IProductInitializer {

	private double initialProductInventoryLevel;

	public ConstantInventoryInitializer(double _initialProductInventoryLevel){
		this.initialProductInventoryLevel = _initialProductInventoryLevel;
	}

	/* (non-Javadoc)
	 * @see com.abcecon.model.IProductInitializer#initializeInventories(java.util.List)
	 */
	public void initializeInventories(List<MarketAgent> _agents) {
		for(int i = 0; i < _agents.size(); i++){
			MarketAgent agent = _agents.get(i);

			// Sets the inventory level to a constant, if this is bigger than the capacity, then we fill to
capacity.
			agent.setInventory(Math.min(initialProductInventoryLevel,agent.capacity) );
		}

	}

}

package com.abcecon.model;

import java.net.URL;
import java.util.List;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

public class DISTANCE {
	/**
	 * Logger for this class
	 */
	private static final Logger logger = Logger.getLogger(DISTANCE.class);

	public DISTANCE(){
		// setup LOGGER
		URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
		PropertyConfigurator.configure(configURL);
		// turn OFF LOGGER
		logger.setLevel(Level.OFF);
```

```
	}


	public static double distanceCalculator(MarketAgent _agent1, MarketAgent _agent2){
		double dist = (_agent1.getX() - _agent2.getX()) * (_agent1.getX() - _agent2.getX()) +
(_agent1.getY() - _agent2.getY()) * (_agent1.getY() - _agent2.getY());
		dist = Math.sqrt(dist);
		return dist;
	}

	/**
	 * Returns the closest MarketAgent.
	 * @param _agents
	 * @param _fromAgent
	 * @return
	 */
	public static MarketAgent findClosestAgent(List<MarketAgent> _agents, MarketAgent _fromAgent){
		double minDist = Double.MAX_VALUE; // to force a distance that we know will never happen
(hopefully)
		MarketAgent closestAgent = null;
		double dist;

		for(int i = 0; i < _agents.size(); i++){
			MarketAgent otherAgent = _agents.get(i);
			if( otherAgent != _fromAgent){
				dist = distanceCalculator(_fromAgent, otherAgent);
				//logger.debug("dist = " + dist);
			}else{
				dist = Double.MAX_VALUE;
			}
			if( dist < minDist){
				minDist = dist;
				closestAgent = otherAgent;
				//logger.debug("minDist = " + minDist + " closestAgent = " +
closestAgent.getId() + " fromAgent = " + _fromAgent.getId());

			}else{
				// ignore, we are looking for a min.
			}
		}


		return closestAgent;

	}



}
```

```
package com.abcecon.model;

import java.util.Date;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class FINANCIAL {

	/**
	 * Returns a Map<Date,Transaction> of all Transactions that are inside a given time period.
	 *
	 * @param _transactionHistory The transactions to search through
	 * @param _startDateOfPeriod The start date of the time period, look for Transactions that occurred on,
or after this Date
	 * @param _endDateOfPeriod The end date of the time period, look for Transactions that occurred on, or
before this Date
	 * @return
	 */
	public static Map<Date,Transaction> getTransactionsInPeriod(Map<Date,Transaction>
_transactionHistory, Date _startDateOfPeriod, Date _endDateOfPeriod){
		Map<Date,Transaction> transactionsInPeriod = new TreeMap<Date,Transaction>();

		// even though we use a TreeMap implementation of the Map interface, we use this method to
be sure, and this allows this method to work for any Map implementation.
		Map<Date,Transaction> transactionsSortedByDate = new
TreeMap<Date,Transaction>(_transactionHistory );


		// First find all Transactions whose Date is after the _startDateOfPeriod and before
_endDateOfPeriod
		Iterator<Date> itDates = transactionsSortedByDate.keySet().iterator();
		while( itDates.hasNext()){
			Date date = itDates.next();
			if ( (date.getTime() >= _startDateOfPeriod.getTime() ) && ( date.getTime() <=
_endDateOfPeriod.getTime() )){
				// then add it to our set of transactions in this time period
				transactionsInPeriod.put(date, transactionsSortedByDate.get(date));
			}else{
				// ignore it, it is out of the time period of interest.
			}
		}

		return transactionsInPeriod;
	}


	/**
	 * Gets the total sales volume inside this time window.
	 *
	 * @param _transactionHistory
	 * @param _startDateOfPeriod
	 * @param _endDateOfPeriod
```

```
	 * @return
	 */
	public static double getTotalSalesInPeriod( Map<Date,Transaction> _transactionHistory, Date
_startDateOfPeriod, Date _endDateOfPeriod ){

		double totalSales = 0.0;
		Map<Date,Transaction> transactions = getTransactionsInPeriod(_transactionHistory,
_startDateOfPeriod,  _endDateOfPeriod);

		Iterator<Transaction> itTransactions = transactions.values().iterator();
		while( itTransactions.hasNext()){
			Transaction trans = itTransactions.next();
			totalSales += trans.getClearingPrice() * trans.getQuantity();
		}

		return totalSales;

	}

}

package com.abcecon.model;


public interface IAskPriceStrategy {


	/**
	 * Implementor returns an ask price based on amount requested.
	 *
	 * @param _quantity
	 * @return
	 */
	public double getAskPriceForQuantity(double _quantity);


	/**
	 * Implementor returns an ask price independent of amount requested.  Can be the same if needed as
above (i.e. just multiply)
	 *
	 * @return
	 */
	public double getAskPricePerUnit();

	/**
	 * Implementor sets the initial ask price.
	 *
	 * @param _initialAskPrice
	 */
	public void setInitialAskPrice(double _initialAskPrice);

}
```

```
package com.abcecon.model;

public interface IMarginalCostCalculator {

	/**
	 * Calculates the additional cost per unit of producing an additional amount of product.
	 *
	 * @param _currentProduction the current level of production
	 * @param _addtionalProduction the amount of extra production
	 * @return
	 */
	public double additionalCost(double _currentProduction, double _addtionalProduction);
}

package com.abcecon.model;

import java.util.List;

public interface IProductInitializer {

	/**
	 * Implementor must assign initial inventory of product to the listed agents.
	 *
	 *
	 * @param _agents
	 */
	public void initializeInventories(List<MarketAgent> _agents);




}

package com.abcecon.model;

public interface ITransactor {

	/**
	 * Implementors try to buy the given amount for the bid price.
	 * @param _amount
	 * @param _price
	 * @return
	 */
	public Transaction buy(double _amount, double _bid);

	/**
	 * Implementors try to sell the given amount for the ask price.
	 * @param _amount
	 * @param _ask
	 * @return
	 */
	public Transaction sell(double _amount, double _ask);
```

```
}

package com.abcecon.model;

public interface IWealthUpdater {

	/**
	 * Implementors must return a <code>BasicAction</code> which can be used for updating the agent's
wealth.
	 * @return
	 */
	public void updateWealth(MarketAgent _agent);

	/**
	 * The frequency to update their wealth in days.
	 *
	 * @return
	 */
	public double getFrequencyInDays();



}

package com.abcecon.model;

public class LinearMarginalCost implements IMarginalCostCalculator {

	/**
	 * The slope of the line that represents the marginal cost calculator.
	 */
	private double slope;




	/**
	 * @param slope
	 */
	public LinearMarginalCost(double slope) {
		super();
		this.slope = slope;
	}




	/**
	 * Returns the slope of the linear marginal cost com.abcecon.model times the amount of extra
production.
	 *
	 */
	public double additionalCost(double _currentProduction, double _additionalProduction) {
```

```
                return slope * _additionalProduction;
        }



        /**
         * @return the slope
         */
        public double getSlope() {
                return slope;
        }



        /**
         * @param slope the slope to set
         */
        public void setSlope(double slope) {
                this.slope = slope;
        }


}

package com.abcecon.model;

import java.net.URL;
import java.util.Comparator;
import java.util.Date;
import java.util.Map;
import java.util.TreeMap;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.engine.CustomProbeable;
import uchicago.src.sim.engine.SimModel;
import uchicago.src.sim.network.DefaultDrawableNode;
import uchicago.src.sim.network.Edge;

import com.abcecon.model.buyers.ISupplierSelector;

public abstract class MarketAgent extends DefaultDrawableNode implements CustomProbeable{

        public static final double DELTA_TIME_STEP = 1.0;

        public static final double NEXT_TIME_TO_CHECK_DAYS = 5.0;

        Logger LOGGER = Logger.getLogger(MarketAgent.class);
```

```
/** For keeping track of agents they have a unique id */
protected String id;

/** Agents have a wealth that changes over time */
private double wealth;


// Here we have our interface for the Strategy Patterns we use.

/** Implementors will update the agents wealth in different ways. */
private IWealthUpdater wealthUpdater;

/** Implementors used to buy or sell */
private ITransactor transactor;

private Map<Date,Transaction> transactionHistory;

/** Reference to the interface of the Simulation that is used to get access to the Schedule */
protected SimModel sim;

/**
 * At what fraction of their capacity does this agent decide to buy more gas?
 */
protected double fractionOfCapacityTriggerToBuy;

/**
 * The size of their tank, or maximum inventory.
 */
protected double capacity;

/**
 * The amount of product the MarketAgent has.
 */
protected double inventory;

/**
 * The frequency in days when a MarketAgent checks its inventory
 */
protected double frequencyDaysCheckInventory;

/**
 * A graph edge from this buyer to the last seller they purchased from;
 */
protected Edge lastTransactionEdge;

/**
 * Interface for different implementations of buying product from sellers strategy.
 */
protected ITransactor buyStrategy;

/**
 * Interface for different implementations of setting the ask price.
```

```
 */
protected IAskPriceStrategy askPriceStrategy;

/**
 * The price the Seller is asking in dollars per unit of product.
 */
protected double askPrice;

/**
 * Strategy pattern for finding a seller. Different agents will use different methods.
 */
protected ISupplierSelector sellerSelector;

/**
 * The number of days that Seller sees whether they should change their asking price.
 */
protected double timePeriodDaysToAdjustAskPrice;

/**
 * Constructor to create an agent.
 */
public MarketAgent(SimModel _simulation, double x, double y){
	super();

	// Set up the Log4j logger
	URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
	PropertyConfigurator.configure(configURL);

	this.sim = _simulation;

	// Sorted map that stores all transactions that this agent participated in, sorted by date.
	this.transactionHistory = new TreeMap<Date,Transaction>();

	// set the position of the agent
	super.setX(x);
	super.setY(y);



}




/**
 * @return the wealth
 */
public double getWealth() {
	return wealth;
}
```

```
/**
 * @param wealth the wealth to set
 */
public void setWealth(double wealth) {
	this.wealth = wealth;
}



/**
 * @return the id
 */
@Override
public String getId() {
	return id;
}



/**
 * @return the wealthUpdater
 */
public IWealthUpdater getWealthUpdater() {
	return wealthUpdater;
}



/**
 * The agent initializes herself on the schedule.
 *
 */
public abstract void initialize();



/**
 * @param wealthUpdater the wealthUpdater to set
 */
public void setWealthUpdater(IWealthUpdater wealthUpdater) {
	this.wealthUpdater = wealthUpdater;
}



/**
 * @return the transactor
 */
public ITransactor getTransactor() {
	return transactor;
```

```
}



/**
 * @param transactor the transactor to set
 */
public void setTransactor(ITransactor transactor) {
	this.transactor = transactor;
}



/**
 * @return the sim
 */
public SimModel getSim() {
	return sim;
}



/**
 * @return the inventory
 */
public double getInventory() {
	return inventory;
}



/**
 * @param inventory the inventory to set
 */
public void setInventory(double gasAmount) {
	this.inventory = gasAmount;
}



/**
 * @return the capacity
 */
```

```
public double getCapacity() {
	return capacity;
}




/**
 * @param capacity the capacity to set
 */
public void setCapacity(double gasCapacity) {
	this.capacity = gasCapacity;
}




/**
 * @return the fractionOfCapacityTriggerToBuy
 */
public double getFractionOfCapacityTriggerToBuy() {
	return fractionOfCapacityTriggerToBuy;
}




/**
 * @param fractionOfCapacityTriggerToBuy the fractionOfCapacityTriggerToBuy to set
 */
public void setFractionOfCapacityTriggerToBuy(double fractionOfCapacityTriggerToBuy) {
	this.fractionOfCapacityTriggerToBuy = fractionOfCapacityTriggerToBuy;
}




/**
 * @return the lastTransactionEdge
 */
public Edge getLastTransactionEdge() {
	return lastTransactionEdge;
}




/**
```

```
 * @return the transactionHistory
 */
public Map<Date, Transaction> getTransactionHistory() {
	return transactionHistory;
}




/**
 * @return the frequencyDaysCheckInventory
 */
public double getFrequencyDaysCheckInventory() {
	return frequencyDaysCheckInventory;
}




/**
 * @param frequencyDaysCheckInventory the frequencyDaysCheckInventory to set
 */
public void setFrequencyDaysCheckInventory(double frequencyDaysCheckInventory) {
	this.frequencyDaysCheckInventory = frequencyDaysCheckInventory;
}




/**
 * @param lastTransactionEdge the lastTransactionEdge to set
 */
public void setLastTransactionEdge(Edge _lastTransactionEdge) {
	this.lastTransactionEdge = _lastTransactionEdge;
}




/**
 * @return the buyStrategy
 */
public ITransactor getBuyStrategy() {
	return buyStrategy;
}
```

```
/**
 * @param buyStrategy the buyStrategy to set
 */
public void setBuyStrategy(ITransactor buyGasStrategy) {
	this.buyStrategy = buyGasStrategy;
}



/**
 * @return the askPriceStrategy
 */
public IAskPriceStrategy getAskPriceStrategy() {
	return askPriceStrategy;
}



/**
 * @param askPriceStrategy the askPriceStrategy to set
 */
public void setAskPriceStrategy(IAskPriceStrategy askPriceStrategy) {
	this.askPriceStrategy = askPriceStrategy;
}



/**
 * @return the askPrice
 */
public double getAskPrice() {
	return this.askPrice;
}



/**
 * @param askPrice the askPrice to set
 */
public void setAskPrice(double askPrice) {
	this.askPrice = askPrice;
}
```

```
	/**
	 * @return the sellerSelector
	 */
	public ISupplierSelector getSellerSelector() {
		return sellerSelector;
	}




	/**
	 * @param sellerSelector the sellerSelector to set
	 */
	public void setSellerSelector(ISupplierSelector sellerSelector) {
		this.sellerSelector = sellerSelector;
	}




	/**
	 * @return the timePeriodDaysToAdjustAskPrice
	 */
	public double getTimePeriodDaysToAdjustAskPrice() {
		return timePeriodDaysToAdjustAskPrice;
	}




	/**
	 * @param timePeriodDaysToAdjustAskPrice the timePeriodDaysToAdjustAskPrice to set
	 */
	public void setTimePeriodDaysToAdjustAskPrice(double checkAskPriceEveryNdays) {
		this.timePeriodDaysToAdjustAskPrice = checkAskPriceEveryNdays;
	}



}


class TransactionTimeComparator implements Comparator<Transaction>{

	/* (non-Javadoc)
	 * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
```

```
		 */
		public int compare(Transaction obj1, Transaction obj2) {
			if( (obj1 instanceof Transaction) && (obj2 instanceof Transaction) ){
				Double time1 = new Double( (obj1).getTime() );
				Double time2 = new Double( (obj2).getTime() );
				return time1.compareTo(time2);

			}else{
				// should not get here
				return 0;
			}

		}
}

package com.abcecon.model;


import java.util.Iterator;

import java.util.List;


import uchicago.src.sim.engine.SimModel;

import uchicago.src.sim.util.Random;

import cern.jet.random.Normal;


import com.abcecon.model.producers.Producer;


public class NormalDistributionProductDistributor implements IProductInitializer {


		private double meanInitialInventory;

		private double stdDevInitialInventory;



		private Normal initialInventoryDistribution;
```

```
	/**
	 *
	 * @param _sim
	 * @param _meanInitialInventory
	 * @param _stdDevInitialInventory
	 */
	public NormalDistributionProductDistributor(SimModel _sim, double _meanInitialInventory, double
_stdDevInitialInventory){

		this.meanInitialInventory = _meanInitialInventory;
		this.stdDevInitialInventory = _stdDevInitialInventory;

		initialInventoryDistribution = new Normal(meanInitialInventory, stdDevInitialInventory,
Random.getGenerator());
	}




	/* (non-Javadoc)
	 * @see com.abcecon.model.IProductInitializer#assignGasToAgents(java.util.List)
	 */
	public void initializeInventories(List<MarketAgent> _agents) {
		Iterator<MarketAgent> itAgents = _agents.iterator();
		while(itAgents.hasNext()){
			MarketAgent agent = itAgents.next();
			// we use this trick to make sure we get into the do while loop at least once, probably a
cleaner way.
			double drawOfProduct = -1.0;
```

```
                    do{
                            drawOfProduct = this.initialInventoryDistribution.nextDouble();
                    }while(drawOfProduct < 0.0);

                    // we need to also get the size of their tanks so we don't give them too much gas...
                    double agentCapacity = agent.getCapacity();

                    agent.setInventory( Math.min(drawOfProduct, agentCapacity )); // should fill the tank if the random draw is > tank size
                }

            }



            /**
             * @return the meanInitialInventory
             */
            public double getMeanInitialInventory() {
                return meanInitialInventory;
            }



            /**
             * @param meanInitialInventory the meanInitialInventory to set
             */
            public void setMeanInitialInventory(double meanInitialInventory) {
```

```
        this.meanInitialInventory = meanInitialInventory;
    }

    /**
     * @return the stdDevInitialInventory
     */
    public double getStdDevInitialInventory() {
        return stdDevInitialInventory;
    }

    /**
     * @param stdDevInitialInventory the stdDevInitialInventory to set
     */
    public void setStdDevInitialInventory(double stdDevInitialInventory) {
        this.stdDevInitialInventory = stdDevInitialInventory;
    }
}

package com.abcecon.model;

import java.util.Comparator;
```

```
public class SimTimeComparator implements Comparator<Double>{

		/* (non-Javadoc)
		 * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
		 */
		public int compare(Double obj1, Double obj2) {
		    if( (obj1 instanceof Double) && (obj2 instanceof Double) ){
				return obj1.compareTo(obj2);
		    }else{
				// should not get here
				return 0;
		    }

		}
}
package com.abcecon.model;

import java.net.URL;



public class SIMUTIL {

		public static final String LOG4FILE = "log4j.properties";

		public static final URL LOG4J_URL = SIMUTIL.class.getResource(LOG4FILE);
```

```
}

package com.abcecon.model;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.SimpleTimeZone;
import java.util.TimeZone;

import com.abcecon.controller.GasSupplyDemandSim;


public class TIME {

        public static final double MILLISECS_PER_SEC = 1000.0;

        public static final double SECONDS_PER_TICK = 1.0 / MILLISECS_PER_SEC;  // Default is milliseconds

        public static final double SECONDS_PER_MINUTE = 60.0;
        public static final double MINUTES_PER_HOUR = 60.0;
        public static final double HOURS_PER_DAY = 24.0;
        public static final double DAYS_PER_WEEK = 7.0;
        public static final double WEEKS_PER_YEAR = 52.0;
```

```
        // note I am avoiding Months.  DAYS_PER_YEAR and DAYS_PER_MONTH, due to the 365.25 days per year, Leap year issues



        /**
         * Returns the number of seconds that correspond to an input DURATION of simulation steps.
         */
        public static double simDurationToSeconds(double _simulationDuration){
            return _simulationDuration * SECONDS_PER_TICK;
        }

        public static double simDurationToMinutes(double _simulationDuration){
            return simDurationToSeconds(_simulationDuration) / SECONDS_PER_MINUTE;
        }

        public static double simDurationToHours(double _simulationDuration){
            return simDurationToMinutes(_simulationDuration) / MINUTES_PER_HOUR;
        }

        public static double simDurationToDays(double _simulationDuration){
            return simDurationToHours(_simulationDuration) / HOURS_PER_DAY;
        }

        public static double simDurationToWeeks(double _simulationDuration){
            return simDurationToDays(_simulationDuration) / DAYS_PER_WEEK;
        }
```

```
public static double simDurationToYear(double _simulationDuration){
    return simDurationToWeeks(_simulationDuration) / WEEKS_PER_YEAR;
}

public static double secondsToSimulationDuration(double _seconds){
    return _seconds * (1.0 / SECONDS_PER_TICK);
}

public static double minutesToSimulationDuration(double _minutes){
    return _minutes * secondsToSimulationDuration(SECONDS_PER_MINUTE);
}

public static double hoursToSimulationDuration(double _hours){
    return _hours * minutesToSimulationDuration(MINUTES_PER_HOUR);
}

public static double daysToSimulationDuration(double _days){
    return _days * hoursToSimulationDuration(HOURS_PER_DAY);
}

public static double weeksToSimulationDuration(double _weeks){
    return _weeks * daysToSimulationDuration(DAYS_PER_WEEK);
}

public static double yearsToSimulationDuration(double _years){
    return _years * weeksToSimulationDuration(WEEKS_PER_YEAR);
```

```
}



/**
 * Creates a utility Calendar for printing out the dates of different simulation times. Use this method for these purposes
 * since the main simulation Calendar is in sync and we don't want to run setTime() on that calendar instance.
 *
 * @param _deltaSimulationTime  The simulation time that you want to set for this utility calendar.
 * @return
 */
public static Calendar createUtilityCalendar(double _simulationTime){
    // get the supported ids for GMT-07:00 (Mountain Standard Time)
    String[] ids = TimeZone.getAvailableIDs(-7 * 60 * 60 * 1000);
    // if no ids were returned, something is wrong. get out.
    if (ids.length == 0)
            System.exit(0);

    // create a Mountain Standard Time time zone
    SimpleTimeZone mdt = new SimpleTimeZone(-8 * 60 * 60 * 1000, ids[0]);

    // set up rules for daylight savings time
    mdt.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);
    mdt.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);
```

```
                    // create a GregorianCalendar with the Mountain Daylight time zone

                    // and the current date and time

                    Calendar utilCalendar = new GregorianCalendar(mdt);


                    utilCalendar.setTime(   new Date((long) _simulationTime +
GasSupplyDemandSim.getStartDate().getTime()) );


                    return utilCalendar;


                }

}

package com.abcecon.model;


import java.net.URL;

import java.util.Date;


import org.apache.log4j.Level;

import org.apache.log4j.Logger;

import org.apache.log4j.PropertyConfigurator;

import org.apache.log4j.helpers.Loader;


import com.abcecon.controller.GasSupplyDemandSim;




public class Transaction {
```

```
static Logger LOGGER = Logger.getLogger(Transaction.class);

/**
 * The simulation time of the transaction.
 */
private double time;

private MarketAgent buyer;

private MarketAgent seller;

/**
 * The agreed upon price where the transaction occurred.
 */
private double clearingPrice;

/**
 * The price that the buyer is willing to pay.
 */
private double bidPrice;

/**
 * The price the seller is willing to sell.
 */
private double askPrice;

/**
```

```
 * The quantity exchanged.
 */
private double quantity;

/**
 * Did the transaction occur? If it did, then the amount was sold for the clearing price.
 *  If it did not, then the buyer and seller did not agree on a price.
 */
private boolean occurred;

public Transaction(){

    //setup the LOGGER
    URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
    PropertyConfigurator.configure(configURL);

    // turn LOGGER off for a while
    LOGGER.setLevel(Level.OFF);

}

/**
 * Create a new Transaction, and STORE it in each parties transaction collections.
 * @param _time
 * @param _buyer
 * @param _seller
```

```
     * @param _amount
     * @param _price
     */
    public Transaction(double _time, MarketAgent _buyer, MarketAgent _seller, double _amount,
double _price){
        this();
        this.time = _time;
        this.buyer = _buyer;
        this.seller = _seller;
        this.quantity = _amount;
        this.clearingPrice = _price;



        // store ourselves in a static container in the Simulation.
        GasSupplyDemandSim.getTransactions().put(new Double(time), this);
        // store ourselves inside the Seller for records
        Date transactionDate = new Date( GasSupplyDemandSim.getStartDate().getTime() + (new
Double(_time)).longValue() );
        if( LOGGER.isDebugEnabled()){
                LOGGER.debug("\t\t adding a transaction at " + transactionDate.toString());
        }
        this.seller.getTransactionHistory().put(transactionDate, this);
    }


    /**
     * @return the time
     */
```

```
public double getTime() {
    return time;
}

/**
 * @param time the time to set
 */
public void setTime(double time) {
    this.time = time;
}

/**
 * @return the buyer
 */
public MarketAgent getBuyer() {
    return buyer;
}

/**
 * @param buyer the buyer to set
 */
public void setBuyer(MarketAgent buyer) {
    this.buyer = buyer;
}

/**
 * @return the seller
```

```
 */
public MarketAgent getSeller() {
    return seller;
}

/**
 * @param seller the seller to set
 */
public void setSeller(MarketAgent seller) {
    this.seller = seller;
}

/**
 * @return the clearingPrice
 */
public double getClearingPrice() {
    return clearingPrice;
}

/**
 * @param clearingPrice the clearingPrice to set
 */
public void setClearingPrice(double clearingPrice) {
    this.clearingPrice = clearingPrice;
}

/**
```

```
     * @return the bidPrice
     */
    public double getBidPrice() {
        return bidPrice;
    }

    /**
     * @param bidPrice the bidPrice to set
     */
    public void setBidPrice(double bidPrice) {
        this.bidPrice = bidPrice;
    }

    /**
     * @return the askPrice
     */
    public double getAskPrice() {
        return askPrice;
    }

    /**
     * @param askPrice the askPrice to set
     */
    public void setAskPrice(double askPrice) {
        this.askPrice = askPrice;
    }
```

```
/**
 * @return the quantity
 */
public double getQuantity() {
    return quantity;
}

/**
 * @param quantity the quantity to set
 */
public void setQuantity(double quantity) {
    this.quantity = quantity;
}

/**
 * @return the occurred
 */
public boolean isOccurred() {
    return occurred;
}

/**
 * @param occurred the occurred to set
 */
public void setOccurred(boolean occurred) {
    this.occurred = occurred;
}
```

```
}

package com.abcecon.model.buyers;


import java.net.URL;

import java.text.DecimalFormat;


import org.apache.log4j.Level;

import org.apache.log4j.Logger;

import org.apache.log4j.PropertyConfigurator;

import org.apache.log4j.helpers.Loader;


import uchicago.src.sim.engine.BasicAction;

import uchicago.src.sim.engine.SimModel;

import uchicago.src.sim.network.Edge;

import uchicago.src.sim.network.EdgeFactory;


import com.abcecon.controller.GasSupplyDemandSim;

import com.abcecon.model.MarketAgent;

import com.abcecon.model.SIMUTIL;

import com.abcecon.model.Transaction;


public class BuyProductAction extends BasicAction {


		Logger LOGGER = Logger.getLogger(BuyProductAction.class);


		private MarketAgent agentBuying;
```

```
	/**
	 * Creates an action that buys product.
	 *
	 * @param _agent the agent doing the buying
	 */
	public BuyProductAction(MarketAgent _agent){
		this.agentBuying = _agent;

		URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
		PropertyConfigurator.configure(configURL);

		// turn LOGGER off for a while
		LOGGER.setLevel(Level.OFF);

	}

	/* (non-Javadoc)
	 * @see uchicago.src.sim.engine.BasicAction#execute()
	 */
	@Override
	public void execute() {
		if( LOGGER.isDebugEnabled()){
				LOGGER.debug("Entering BuyProductAction.execute() method for agent = " +
this.agentBuying.getId());
		}

		// check how much product we have relative to our trigger level to buy
```

```
                if( agentBuying.getInventory() < ( agentBuying.getCapacity() *
agentBuying.getFractionOfCapacityTriggerToBuy() )){



                        if( LOGGER.isDebugEnabled()){



                                LOGGER.debug("Current time is " +
((GasSupplyDemandSim)agentBuying.getSim()).getCurrentDate().toString() +
                                                ", " + agentBuying.getId() + " needs to buy gas at time " +
GasSupplyDemandSim.getCalendar().getTime().toString());

                        }



                        // find a seller

                        ISupplierSelector sellerSelector = agentBuying.getSellerSelector();

                        if( sellerSelector != null){

                                MarketAgent seller = sellerSelector.findSupplier();



                                // how much do I need to buy?

                                double amountToFillCapacity = agentBuying.getCapacity() -
agentBuying.getInventory();



                                // how much is the seller selling for? At this point we have decided to buy it
from this Supplier.

                                double pricePaidPerUnit = seller.getAskPrice();



                                SimModel sim = seller.getSim();
```

```
                    // create a new Transaction(double _time, MarketAgent _buyer, MarketAgent
_seller, double _amount, double _price){
                    Transaction transaction = new Transaction(sim.getSchedule().getCurrentTime(),
agentBuying, seller, amountToFillCapacity, pricePaidPerUnit );


                    // ADD edge on graph, delete the old one if it exists
                    if( agentBuying.getLastTransactionEdge() != null){
                        Edge lastEdge = agentBuying.getLastTransactionEdge();
                        agentBuying.removeOutEdge(lastEdge);
                        lastEdge = null;

        agentBuying.setLastTransactionEdge(EdgeFactory.createDrawableEdge(agentBuying, seller));
                    }else{

        agentBuying.setLastTransactionEdge(EdgeFactory.createDrawableEdge(agentBuying, seller));
                    }


                    // update the display
                    ((GasSupplyDemandSim) sim).getSpatialDisplaySurface().updateDisplay();


                    if( LOGGER.isDebugEnabled()){
                        double inventoryBeforeBuying = agentBuying.getInventory();
                        double amountOfProductBought = amountToFillCapacity;
                        double inventoryAfterBuying = agentBuying.getInventory() +
amountToFillCapacity; // should really check using a new call to agentBuying.getInventory() after purchase is
made.
                        DecimalFormat decimalFormatter = new DecimalFormat("###,###.##"
);
                        StringBuffer buf = new
StringBuffer(((GasSupplyDemandSim)agentBuying.getSim()).getCurrentDate().toString() + ": " + agentBuying.getId()
+ " (pre-purchase inventory, purchase amt, post inventory) = ( ");
```

```
                                        buf.append(decimalFormatter.format(inventoryBeforeBuying) + ", " +
decimalFormatter.format(amountOfProductBought) + ", " + decimalFormatter.format(inventoryAfterBuying) +" )");


                                        LOGGER.debug(buf.toString());

                                }


                                //add product to the buyer's inventory level

                                agentBuying.setInventory(agentBuying.getInventory() + amountToFillCapacity
);

                                // remove product from the seller's inventory level

                                seller.setInventory(seller.getInventory() - amountToFillCapacity);


                                if( LOGGER.isDebugEnabled()){

                                        DecimalFormat decimalFormatter = new DecimalFormat("###,###.##"
);

                LOGGER.debug(((GasSupplyDemandSim)agentBuying.getSim()).getCurrentDate().toString() + ": " +
agentBuying.getId() + " has inventory after buying = " + decimalFormatter.format(agentBuying.getInventory()));

                                }


                                // take money out of buyer's bank account

                                agentBuying.setWealth(agentBuying.getWealth() - amountToFillCapacity *
pricePaidPerUnit);


                                // add money to the seller's bank account

                                seller.setWealth(seller.getWealth() + amountToFillCapacity * pricePaidPerUnit);


                        }else{

                                LOGGER.fatal("No implementor of ISupplierSelector assigned to " +
agentBuying.getId());
```

```
                    }
                }
            }
}
package com.abcecon.model.buyers;

import java.util.Collections;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.sellers.PriceComparator;


public class CheapestSellerSelector implements ISupplierSelector {


        private GasSupplyDemandSim sim;


        public CheapestSellerSelector(GasSupplyDemandSim _sim){
            this.sim = _sim;

        }

        /* (non-Javadoc)
         * @see com.abcecon.model.ISellerSelector#findSeller()
```

```
		 */
		public MarketAgent findSupplier() {
			Collections.sort(sim.getGasSellers(), new PriceComparator());
			// forgot if this sorts high to low or vice versa
			MarketAgent seller0 = sim.getGasSellers().get(0);
			MarketAgent sellerN = sim.getGasSellers().get(sim.getGasSellers().size() - 1);
			if( seller0.getAskPrice() < sellerN.getAskPrice()){
				return seller0;
			}else{
				return sellerN;
			}

		}
}

package com.abcecon.model.buyers;

import uchicago.src.sim.engine.SimModel;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.DISTANCE;
import com.abcecon.model.MarketAgent;

public class ClosestSellerSelector implements ISupplierSelector {

		private SimModel sim;
```

```
private MarketAgent buyer;


/**
 * Constructor for this strategy.
 * @param _sim
 * @param _buyer
 */
public ClosestSellerSelector(SimModel _sim, MarketAgent _buyer){
    this.sim = _sim;
    this.buyer = _buyer;
}




//ED_2_SHANE start
// Shane, I removed a constructor that you had here, that I did not, this was the bug...
//ED_2_SHANE end


/*
 * Finds the closest seller.
 *
 * (non-Javadoc)
 * @see com.abcecon.model.ISellerSelector#findSeller()
 */
public MarketAgent findSupplier() {
```

```
                //ED_TODO JUnit test this thing...


                double minDist = 1.0E20; // initial value is huge to start things out (lame, I think).
                double dist = - 1.0;
                MarketAgent closestSeller = null;
                for(int i = 0; i <  ((GasSupplyDemandSim)sim).getGasSellers().size(); i++){
                        MarketAgent seller = ((GasSupplyDemandSim)sim).getGasSellers().get(i);
                        if( !seller.equals(buyer)){
                                dist = DISTANCE.distanceCalculator(seller, buyer);
                                if( dist < minDist){
                                        minDist = dist;
                                        closestSeller = seller;
                                }
                        }
                }
                return closestSeller;
        }
}
package com.abcecon.model.buyers;


import java.net.URL;
import java.text.DecimalFormat;


import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;
```

```
import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.CheckInventoryAction;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.SIMUTIL;

public class ConsumeProductAction extends BasicAction {

		Logger LOGGER = Logger.getLogger(ConsumeProductAction.class);

		private Consumer consumer;
		private double periodOfConsumptionInDays;

		/**
		 * Set up a consumption of product equal to the period of consumption in days, times the daily consumption rate.
		 * @param _consumer
		 * @param _periodOfConsumptionInDays
		 */
		public ConsumeProductAction(Consumer _consumer, double _periodOfConsumptionInDays){
			consumer = _consumer;

			URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
			PropertyConfigurator.configure(configURL);
```

```
        // turn LOGGER off for a while
        LOGGER.setLevel(Level.OFF);



        this.periodOfConsumptionInDays = _periodOfConsumptionInDays;

    }

    /* (non-Javadoc)
     * @see uchicago.src.sim.engine.BasicAction#execute()
     */
    @Override
    public void execute() {
        // consume an amount of daily rate * number of days
        if( consumer.getInventory() > (consumer.getCapacity() *
consumer.getFractionOfCapacityTriggerToBuy() ) ){



            double amountConsumed = consumer.getDailyConsumptionRate() *
periodOfConsumptionInDays;
            if(LOGGER.isDebugEnabled()){
                DecimalFormat formatter = new DecimalFormat("#,###.##");

    LOGGER.debug(((GasSupplyDemandSim)consumer.getSim()).getCurrentDate().toString() +
                                    ": " + consumer.getId() + " has consumed " +
formatter.format(amountConsumed)+ " units of product");
            }
            consumer.setInventory( consumer.getInventory() - amountConsumed );
        }else{
```

```
                        // then call the CheckInventoryAction, this should trigger them to buy gas
                        Schedule schedule = consumer.getSim().getSchedule();
                        schedule.scheduleActionAt(schedule.getCurrentTime() +
MarketAgent.DELTA_TIME_STEP, new CheckInventoryAction(consumer));
                    }
                }

}

package com.abcecon.model.buyers;

import java.awt.Color;
import java.awt.Image;
import java.awt.geom.Rectangle2D;
import java.net.URL;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;

import javax.swing.ImageIcon;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;
```

```
import uchicago.src.sim.engine.BasicAction;

import uchicago.src.sim.engine.Schedule;

import uchicago.src.sim.engine.SimModel;

import uchicago.src.sim.gui.SimGraphics;


import com.abcecon.model.CheckInventoryAction;

import com.abcecon.model.MarketAgent;

import com.abcecon.model.SIMUTIL;

import com.abcecon.model.TIME;

import com.abcecon.model.Transaction;


public class Consumer extends MarketAgent {


		Logger LOGGER = Logger.getLogger(Consumer.class);


		/** A static counter to help us create a unique id of each agent. The reason we use "static" is that this means that this number is the SAME for ALL instances

		 * of Agents, that is it resides inside the Class instead of inside each instance of an MarketAgent Class.

		 */

		private static int counter;



		private static Image carPicture;


		private List<Transaction> buyTransactions;


		/** Implementaors adjust consumption behaviors in different ways */
```

```
private IAdjustConsumptionBehavior adjustConsumptionStrategy;

/**
 * This is the rate of gas consumption per day
 */
private double dailyConsumptionRate ;

/**
 * The annual salary of the consumer.
 */
private double weeklyPaycheck;

/**
 * @param _simulation
 * @param x
 * @param y
 */
public Consumer(SimModel _simulation, double x, double y) {
    super(_simulation, x, y);


    URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
    PropertyConfigurator.configure(configURL);

    // turn Logger off
    this.LOGGER.setLevel(Level.OFF);
```

```
        // increment the count and assign the unique id
        super.id = "Consumer_" + counter;

        this.buyTransactions = new ArrayList<Transaction>();

        loadCarImage();
        this.setNodeLabel(this.id);


        Consumer.counter++; // increment the counter. Note how I call the static field here.
    }



    /**
     * @return the weeklyPaycheck
     */
    public double getWeeklyPaycheck() {
        return weeklyPaycheck;
    }



    /**
     * @param weeklyPaycheck the weeklyPaycheck to set
     */
```

```
public void setWeeklyPaycheck(double annualWealth) {
    this.weeklyPaycheck = annualWealth;
}

/**
 * @return the dailyConsumptionRate in Gallons / Day
 */
public double getDailyConsumptionRate() {
    return dailyConsumptionRate ;
}

/**
 * @param dailyConsumptionRate the dailyConsumptionRate to set
 */
public void setDailyConsumptionRate(double dailyGasConsumptionRate) {
    this.dailyConsumptionRate = dailyGasConsumptionRate;
}

/**
 * @return the adjustConsumptionStrategy
 */
public IAdjustConsumptionBehavior getAdjustConsumptionStrategy() {
```

```
        return adjustConsumptionStrategy;
    }

    /**
     * @param adjustConsumptionStrategy the adjustConsumptionStrategy to set
     */
    public void setAdjustConsumptionStrategy(
            IAdjustConsumptionBehavior adjustConsumptionStrategy) {
        this.adjustConsumptionStrategy = adjustConsumptionStrategy;
    }

    /**
     *
     * The initialize method is where we schedule the initial actions of the Consumer agent.  This agent can reschedule future actions
     * and also change actions if it wants.  It is autonomously scheduling itself.
     *
     *
     */
    @Override
    public void initialize(){
        Schedule scheduler = this.getSim().getSchedule();

        // UPDATE WEALTH: Call the implementor of IWealthUpdater and update the Wealth of this agent.
        if( this.getWealthUpdater() != null){
            this.getWealthUpdater().updateWealth(this);
        }else{
```

```
                    LOGGER.error("WealthUpdater is null for " + this.getId());
                }

                // CHECK INVENTORY AND BUY PRODUCT: add the check inventory action to the schedule, it should reschedule itself inside the CheckInventoryAction class.
                double simTime = TIME.daysToSimulationDuration( this.getFrequencyDaysCheckInventory() );
                scheduler.scheduleActionAt(simTime, new CheckInventoryAction(this)); // Note that CheckInventoryAction can also BUY product.

                // CONSUMER PRODUCT: kick off the consumption of product, it is consuming at a daily rate
                scheduler.scheduleActionAtInterval(TIME.daysToSimulationDuration( 1.0 ), new ConsumeProductAction(this, 1.0));
            }

            /**
             * @return the buyTransactions
             */
            public List<Transaction> getBuyTransactions() {
                return buyTransactions;
            }

            private static void loadCarImage() {
                if (carPicture == null) {
                    // would like to change this to have a test to see if the agent is a seller or not, and then change color accordingly... but it is static.
                    java.net.URL carPicURL = MarketAgent.class.getResource("car.gif");
                    carPicture = new ImageIcon(carPicURL).getImage();
                }
            }
```

```
        @Override
        public void draw(SimGraphics g) {
            // draw the agent's picture
            g.drawImage(carPicture);

            // grab the width of the picture
            int width = carPicture.getWidth(null);

            g.setFont(super.getFont());

            // get the size of the node's text
            Rectangle2D bounds = g.getStringBounds(this.getNodeLabel());

            // set the graphics to draw the text above the label
            // the x coordinate is relative to the upper left corner of the image
            // so the coordinates are shifted to account for that
            g.setDrawingCoordinates((float) (this.getX() + width / 2.0 - bounds.getWidth() / 2.0),(float)
(this.getY() - bounds.getHeight() - 2), 0f);

            // draw the label
            g.drawString(getNodeLabel(), Color.BLACK);
        }

        public String[] getProbedProperties() {
            return new String[] {"id",
"wealth","capacity","inventory","dailyConsumptionRate","fractionOfCapacityTriggerToBuy"};
```

```
        }

        @Override
        public String toString(){
            DecimalFormat decimalFormatter = new DecimalFormat("###,###.##");
            NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();

            StringBuffer buf = new StringBuffer("Consumer:");
            buf.append( "\n\t id = " + this.id );
            buf.append( "\n\t capacity = " + decimalFormatter.format( this.capacity ) );
            buf.append("\n\t inventory = " + decimalFormatter.format( this.inventory) );
            buf.append("\n\t fractionOfCapacityTriggerToBuy = " + decimalFormatter.format(
this.fractionOfCapacityTriggerToBuy ) );
            buf.append("\n\t wealth = " + currencyFormatter.format( this.getWealth() ) );
            buf.append("\n\t daily consumption rate = " + decimalFormatter.format( dailyConsumptionRate)
);

            return buf.toString();
        }


        class ConsumeProduct extends BasicAction{

            private MarketAgent consumer;

            public ConsumeProduct(MarketAgent _consumer){
                    this.consumer = _consumer;
            }
```

```
		/* (non-Javadoc)
		 * @see uchicago.src.sim.engine.BasicAction#execute()
		 */
		@Override
		public void execute() {
			// first see if we have enough product
			if( consumer.getInventory() > 0.0 ){
				// then consume
			}else{
				Schedule scheduler = consumer.getSim().getSchedule();
				double DELTA = 1.2; // just move it off of now
				double curTime = scheduler.getCurrentTime();
				scheduler.scheduleActionAt(curTime + DELTA, new
BuyProductAction(consumer));
			}

		}

	}

}
```

```
package com.abcecon.model.buyers;

import java.util.Collections;
import java.util.Date;

import com.abcecon.model.Transaction;

public class ConsumerBehavior_BudgetBased implements IAdjustConsumptionBehavior {

	private double budgetAsFractionOfWeeklyPaycheck;

	public ConsumerBehavior_BudgetBased(double _budgetAsFractionOfWeeklyPaycheck){
	    this.budgetAsFractionOfWeeklyPaycheck = _budgetAsFractionOfWeeklyPaycheck;
	}

	/* (non-Javadoc)
	 * @see com.abcecon.model.buyers.IAdjustConsumptionBehavior#getNewConsumptionRate(com.abcecon.model.MarketAgent)
	 */
	public double getNewConsumptionRate(Consumer _consumer) {

	    // get the agent's weekly paycheck
	    double weeklyPaycheck = _consumer.getWeeklyPaycheck();

	    // amount to spend on gas per week
	    double weeklyGasBudget = this.budgetAsFractionOfWeeklyPaycheck * weeklyPaycheck;
```

```
                // get the gas price they paid last time per gallon of gas.
                Date lastTransactionDate = Collections.max( _consumer.getTransactionHistory().keySet() );

                // get the amount of this last transition
                Transaction lastTransaction = _consumer.getTransactionHistory().get(lastTransactionDate);

                lastTransaction.getClearingPrice();



                return 0;
            }

}
package com.abcecon.model.buyers;


public interface IAdjustConsumptionBehavior {


            /**
             * Implementors must return a new consumption rate.
             *
             * @param _agent
             * @return
             */
            public double getNewConsumptionRate(Consumer _agent);
```

```
}
package com.abcecon.model.buyers;

import java.util.List;

import com.abcecon.model.MarketAgent;

public interface IConsumptionRateAssigner {

		public void assignDailyConsumptionRate(List<MarketAgent> _consumers);


}
package com.abcecon.model.buyers;

import com.abcecon.model.MarketAgent;


public interface ISupplierSelector {

		/**
		 * Implementors return a supplier of product.
		 *
		 * @return
		 */
		public MarketAgent findSupplier();
```

```
}

package com.abcecon.model.buyers;


import java.util.Iterator;

import java.util.List;


import com.abcecon.model.MarketAgent;


import uchicago.src.sim.engine.SimModelImpl;

import uchicago.src.sim.util.Random;

import cern.jet.random.Normal;



public class NormalDistributionConsumptionRateAssigner implements IConsumptionRateAssigner {


		private double meanDailyConsumption;

		private double stdDevDailyConsumption;


		private Normal consumptionRateDistribution;



		public NormalDistributionConsumptionRateAssigner(SimModelImpl _sim, double
_meanDailyConsumption, double _stdDevDailyConsumption){


			this.meanDailyConsumption = _meanDailyConsumption;

			this.stdDevDailyConsumption = _stdDevDailyConsumption;
```

```
                this.consumptionRateDistribution = new
Normal(meanDailyConsumption,stdDevDailyConsumption,Random.getGenerator());
        }

        /* (non-Javadoc)
         * @see
com.abcecon.model.IGasConsumptionRateAssigner#assignDailyGasConsumptionRate(java.util.List)
         */
        public void assignDailyConsumptionRate(List<MarketAgent> _agents) {

            Iterator <MarketAgent> itBuyers = _agents.iterator();

            while (itBuyers.hasNext()){
                Consumer gasBuyer = (Consumer) itBuyers.next();
                double rate = - 1.0;
                do {
                    rate = this.consumptionRateDistribution.nextDouble();
                }while(rate <= 0);

                gasBuyer.setDailyConsumptionRate(rate);
            }

        }

}
```

```
package com.abcecon.model.buyers;



import uchicago.src.sim.util.Random;

import cern.jet.random.Uniform;


import com.abcecon.controller.GasSupplyDemandSim;

import com.abcecon.model.MarketAgent;



public class RandomSellerSelector implements ISupplierSelector {



		private GasSupplyDemandSim sim;


		public RandomSellerSelector(GasSupplyDemandSim _sim){

		    this.sim = _sim;


		}



		/*

		 * Finds a random seller.

		 *

		 * (non-Javadoc)
```

```
     * @see com.abcecon.model.ISellerSelector#findSeller()
     */

    /**
     * @param gasSupplyDemandSim
     * @param findRandomSellerBuyer
     */
    public RandomSellerSelector(GasSupplyDemandSim gasSupplyDemandSim, MarketAgent
findRandomSellerBuyer) {

        //ED_2_SHANE start
        // you did not set the local class variable of sim to the value passed into this constructor
        this.sim = gasSupplyDemandSim;
    }


    public MarketAgent findSupplier() {

        int sellersTotal = sim.getSellersNumberOf();

        //ED_2_SHANE
        //use the Repast Uniform distribution to pick random number that is in range of
[0,numberSellers-1]
        Uniform uniformDistribution = new Uniform(0, sellersTotal -1 , Random.getGenerator());

        int sellerNumber = uniformDistribution.nextInt();

        MarketAgent seller = ((GasSupplyDemandSim)sim).getGasSellers().get(sellerNumber);
```

```
                return seller;
            }
}

package com.abcecon.model.buyers;

import java.net.URL;
import java.text.NumberFormat;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.engine.BasicAction;

import com.abcecon.model.IWealthUpdater;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.SIMUTIL;
import com.abcecon.model.TIME;

public class WeeklyWealthUpdater implements IWealthUpdater {

        Logger LOGGER = Logger.getLogger(WeeklyWealthUpdater.class);


        /**
```

```
     * Create an new implementation instance that will give out a weekly paycheck.
     */
    public WeeklyWealthUpdater(){
        URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
        PropertyConfigurator.configure(configURL);
    }


    /* (non-Javadoc)
     * @see com.abcecon.model.IWealthUpdater#updateWealth(com.abcecon.model.MarketAgent)
     */
    public void updateWealth(MarketAgent _agent) {
        _agent.getSim().getSchedule().scheduleActionAtInterval( TIME.weeksToSimulationDuration( 1.0
), new WealthUpdateAction(_agent) );


    }

    /* (non-Javadoc)
     * @see com.abcecon.model.IWealthUpdater#getFrequency()
     */
    public double getFrequencyInDays() {
        return TIME.DAYS_PER_WEEK;
    }
```

```
class WealthUpdateAction extends BasicAction{

    private MarketAgent agent;

    /**
     * This is found by taking the agents initial wealth and dividing by 52 weeks.  We will do this at start up time of the simulation.
     */
    double averageWeeklyPaycheck;
    boolean paycheckSizeCalculated = false;
    double initialWealth;

    /** Useful built in formatting utility of Java I will use */
    NumberFormat dollarFormat = NumberFormat.getCurrencyInstance();

    public WealthUpdateAction(MarketAgent _agent){
        agent = _agent;
    }

    /* (non-Javadoc)
     * @see uchicago.src.sim.engine.BasicAction#execute()
     */
    @Override
    public void execute() {

        double wealth = agent.getWealth();
```

```
                        // this should only run once, the first time to set the paycheck sizes.
                        if( !paycheckSizeCalculated ){
                                this.initialWealth = wealth;
                                this.averageWeeklyPaycheck = initialWealth / TIME.WEEKS_PER_YEAR ;

                                // I had to put this here since in order to use ConsumerBehavior_BudgetBased,
EPM
                                ((Consumer) agent).setWeeklyPaycheck(averageWeeklyPaycheck);

                                this.paycheckSizeCalculated = true;
                        }else{
                                // then add wealth to agent
                                double curWealth = wealth;
                                agent.setWealth(wealth + averageWeeklyPaycheck);

                                if( LOGGER.isDebugEnabled()){
//                                      NumberFormat formatter = NumberFormat.getCurrencyInstance();
//                                      GasSupplyDemandSim sim = (GasSupplyDemandSim)agent.getSim();
//                                      LOGGER.debug(sim.getCurrentDate().toString() + ": " + agent.getId() +
//                                                      " (current wealth, paycheck, new wealth) = (" +
formatter.format(curWealth) +
//                                                      ", " + formatter.format(averageWeeklyPaycheck) + ",
" + formatter.format(agent.getWealth()) +")" );
                                }
                        }

                }
```

```
                }


}
package com.abcecon.model.producers;

import java.awt.Color;
import java.awt.Image;
import java.awt.geom.Rectangle2D;

import javax.swing.ImageIcon;

import com.abcecon.model.MarketAgent;
import com.abcecon.model.ITransactor;

import uchicago.src.sim.engine.SimModel;
import uchicago.src.sim.gui.SimGraphics;

/**
 * An agent that produces product and sells it to the Seller level normally.
 *
 * @author mackerrow  Jan 26, 2009   6:57:03 PM
 *
 */
public class Producer extends MarketAgent {
```

```
        /** A static counter to help us create a unique id of each agent. The reason we use "static" is that
this means that this number is the SAME for ALL instances
         * of GasProducers, that is it resides inside the Class instead of inside each instance of an Producer
Class.
         */
        private static int counter = 0;

        private double inventory;

        /**
         * Fixed costs are independent of whether the seller is selling product or not.
         */
        private double fixedCosts;

        /**
         * Costs that are correlated with the amount of product that is sold.
         */
        private double variableCosts;


        private ITransactor sellToProducersStrategy;


        private static Image gasProducerImage;


        /**
```

```
     * @param _simulation
     * @param x
     * @param y
     */
    public Producer(SimModel _simulation, double x, double y) {
        super(_simulation, x, y);
        // increment the count and assign the unique id

        super.id = "GasProducer_" + counter; // Java Strings are smart enough now to create a String out of a integer combined with a String.

        loadGasProducerImage();
        this.setNodeLabel(this.id);

        Producer.counter++; // increment the counter. Note how I call the static field here.

    }

    /* (non-Javadoc)
     * @see com.abcecon.model.MarketAgent#initialize()
     */
    @Override
```

```
public void initialize() {
    // TODO Auto-generated method stub

}

/**
 * @return the inventory
 */
@Override
public double getInventory() {
    return inventory;
}

/**
 * @param inventory the inventory to set
 */
@Override
```

```
public void setInventory(double inventory) {
    this.inventory = inventory;
}

/**
 * @return the fixedCosts
 */
public double getFixedCosts() {
    return fixedCosts;
}

/**
 * @param fixedCosts the fixedCosts to set
 */
public void setFixedCosts(double fixedCosts) {
    this.fixedCosts = fixedCosts;
}
```

```
/**
 * @return the variableCosts
 */
public double getVariableCosts() {
    return variableCosts;
}

/**
 * @param variableCosts the variableCosts to set
 */
public void setVariableCosts(double variableCosts) {
    this.variableCosts = variableCosts;
}
```

```
/**
 * @return the sellToProducersStrategy
 */
public ITransactor getSellToProducersStrategy() {
    return sellToProducersStrategy;
}

/**
 * @param sellToProducersStrategy the sellToProducersStrategy to set
 */
public void setSellToProducersStrategy(ITransactor sellToProducersStrategy) {
    this.sellToProducersStrategy = sellToProducersStrategy;
}

/**
```

```
     * @return the gasProducerImage
     */
    public static Image getGasProducerImage() {
        return gasProducerImage;
    }




    private static void loadGasProducerImage() {
        if (gasProducerImage == null) {
                // would like to change this to have a test to see if the agent is a seller or not, and then change color accordingly... but it is static.
                java.net.URL producerPicURL = MarketAgent.class.getResource("oilPump.gif");
                gasProducerImage = new ImageIcon(producerPicURL).getImage();
        }
    }



    @Override
    public void draw(SimGraphics g) {
        // draw the agent's picture
        g.drawImage(gasProducerImage);
```

```
            // grab the width of the picture
            int width = gasProducerImage.getWidth(null);

            g.setFont(super.getFont());

            // get the size of the node's text
            Rectangle2D bounds = g.getStringBounds(this.getNodeLabel());

            // set the graphics to draw the text above the label
            // the x coordinate is relative to the upper left corner of the image
            // so the coordinates are shifted to account for that
            g.setDrawingCoordinates((float) (this.getX() + width / 2.0 - bounds.getWidth() / 2.0),(float)
(this.getY() - bounds.getHeight() - 2), 0f);

            // draw the label
            g.drawString(getNodeLabel(), Color.BLACK);
        }

        public String[] getProbedProperties() {
            return new String[] {"id", "wealth"};
        }
```

```
}
package com.abcecon.model.sellers;

import java.util.Comparator;

import com.abcecon.model.MarketAgent;



public class PriceComparator implements Comparator<MarketAgent>{

		/* (non-Javadoc)
		 * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
		 */
		public int compare(MarketAgent obj1, MarketAgent obj2) {
		    if( (obj1 instanceof MarketAgent) && (obj2 instanceof MarketAgent) ){
				double askPrice1 = (obj1).getAskPrice();
				double askPrice2 = (obj2).getAskPrice();
				if( askPrice1 > askPrice2){
					return 1;
				}else if( askPrice1 < askPrice2 ){
					return -1;
				}else{
					return 0;
				}


		    }else{
```

```
                    // should not get here
                    return 0;
                }

            }
}
package com.abcecon.model.sellers;

import java.net.URL;
import java.text.NumberFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.List;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.DISTANCE;
import com.abcecon.model.FINANCIAL;
import com.abcecon.model.IAskPriceStrategy;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.SIMUTIL;
import com.abcecon.model.TIME;
```

```
public class SellerCompetitivePriceTakerStrategy implements IAskPriceStrategy {

		/**
		 * Logger for this class
		 */
		private static final Logger logger = Logger.getLogger(SellerCompetitivePriceTakerStrategy.class);



		private MarketAgent seller;


		//private int timePeriodToMeasureSalesPerformanceInDays = 7;


		/**
		 * The change in price that the Seller will move their price each price move, either up or down.
		 */
		private double deltaPriceChangeInDollars = 0.01;




		public SellerCompetitivePriceTakerStrategy(MarketAgent _seller){

		    this.seller = _seller;

		    URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
		    PropertyConfigurator.configure(configURL);

		}
```

```
	/* (non-Javadoc)
	 * @see com.abcecon.model.IAskPriceStrategy#getAskPrice()
	 */
	public double getAskPricePerUnit() {
		double askPrice = seller.getAskPrice();

		// find the price of the closest Seller (competition)
		List<MarketAgent> sellers = ((GasSupplyDemandSim)seller.getSim()).getGasSellers();
		MarketAgent closestSeller = DISTANCE.findClosestAgent(sellers, seller);

		// compare our sales rate last time period, to the time period before. If the last time period had an increase in sales, then we raise the price, otherwise lower it
		// to be below the closest competition.
		Date currentDate = ((GasSupplyDemandSim)seller.getSim()).getCurrentDate(); // This call sets the calendar to the current sim time

		Calendar cal = TIME.createUtilityCalendar(seller.getSim().getSchedule().getCurrentTime());

		// find the Date one time period ago
		cal.roll(Calendar.DAY_OF_YEAR, new Double(-seller.getTimePeriodDaysToAdjustAskPrice()).intValue());
		Date dateAtStartOfPeriodOneTimePeriodBack = cal.getTime();

		// find the Date two time periods ago
		cal.roll(Calendar.DAY_OF_YEAR, new Double(-seller.getTimePeriodDaysToAdjustAskPrice()).intValue() ); // yep, just roll the calendar back one more time
		Date dateAtStartOfPeriodTwoTimePeriodsBack = cal.getTime();
```

```
		if (logger.isInfoEnabled()) {

			logger.debug("CurrentDate from sim: " + currentDate.toString() + ", From calendar: " +
GasSupplyDemandSim.getCalendar().getTime() ); //$NON-NLS-1$

			logger.debug("Date one period back: "
+dateAtStartOfPeriodOneTimePeriodBack.toString() +", Date two periods back: " +
dateAtStartOfPeriodTwoTimePeriodsBack.toString() );

		}




		double salesLastPeriod = FINANCIAL.getTotalSalesInPeriod(seller.getTransactionHistory(),
dateAtStartOfPeriodOneTimePeriodBack, currentDate);

		double salesPreviousPeriod = FINANCIAL.getTotalSalesInPeriod(seller.getTransactionHistory(),
dateAtStartOfPeriodTwoTimePeriodsBack, dateAtStartOfPeriodOneTimePeriodBack);



		if (logger.isInfoEnabled()) {

			NumberFormat formatter = NumberFormat.getCurrencyInstance();

			logger.debug("Sales previous period = " +formatter.format(salesPreviousPeriod));
//$NON-NLS-1$

			logger.debug("Sales last period = " +formatter.format(salesLastPeriod)); //$NON-NLS-1$

		}


		if( salesPreviousPeriod != 0.0 && salesPreviousPeriod != 0.0){

			if(  salesLastPeriod <= salesPreviousPeriod ){

				// Sales are dropping, so then we lower our ask price

				double competitorCurrentAskPrice = closestSeller.getAskPrice();

				if( competitorCurrentAskPrice <= askPrice){

					askPrice = competitorCurrentAskPrice -
this.deltaPriceChangeInDollars;
```

```
                }else{
                        // then just drop our price one notch
                        askPrice = askPrice - this.deltaPriceChangeInDollars;
                }

        }else{
                // then we raise our ask price relative to our previous price
                askPrice = askPrice + this.deltaPriceChangeInDollars;
        }

    }


    return askPrice;
}

/* (non-Javadoc)
 * @see com.abcecon.model.IAskPriceStrategy#getAskPrice(double)
 */
public double getAskPriceForQuantity(double _quantity) {
    // no quantity discount
    return _quantity * this.getAskPricePerUnit();
}
```

```
/* (non-Javadoc)
 * @see com.abcecon.model.IAskPriceStrategy#setInitialAskPrice(double)
 */
public void setInitialAskPrice(double askPrice) {
     (seller).setAskPrice(askPrice);

}



/**
 * The price the Seller will change their ask price every price move.
 *
 * @return the deltaPriceChangeInDollars
 */
public double getDeltaPriceChangeInDollars() {
     return deltaPriceChangeInDollars;
}



/**
 *  The price the Seller will change their ask price every price move.
 *
 * @param deltaPriceChangeInDollars the deltaPriceChangeInDollars to set
 */
public void setDeltaPriceChangeInDollars(double deltaPriceChangeInDollars) {
```

```
            this.deltaPriceChangeInDollars = deltaPriceChangeInDollars;
        }
}
package com.abcecon.model.sellers;

import java.net.URL;
import java.text.DecimalFormat;
import java.text.NumberFormat;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.SIMUTIL;
import com.abcecon.model.TIME;
```

```
public class ChangeAskPriceAction extends BasicAction {
		/**
		 * Logger for this class
		 */
		private static final Logger logger = Logger.getLogger(ChangeAskPriceAction.class);



		private MarketAgent seller;

		private Schedule schedule;

		public ChangeAskPriceAction(MarketAgent _seller){
			super();

			URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
			PropertyConfigurator.configure(configURL);

			this.seller = _seller;
			this.schedule = _seller.getSim().getSchedule();

		}

		/* (non-Javadoc)
		 * @see uchicago.src.sim.engine.BasicAction#execute()
		 */
		@Override
```

```
        public void execute() {
            double oldAskPrice = seller.getAskPrice();
            // get the price change mechanism
            double newAskPrice = seller.getAskPriceStrategy().getAskPricePerUnit();
            seller.setAskPrice(newAskPrice);

            // reschedule ourselves to check the price again in the future
            double nextTime =  ((GasSupplyDemandSim) seller.getSim()).getSchedule().getCurrentTime() +
TIME.daysToSimulationDuration(seller.getTimePeriodDaysToAdjustAskPrice());

            if (logger.isDebugEnabled()) {
                GasSupplyDemandSim.getCalendar().setTimeInMillis( (new
Double(nextTime)).longValue());
                DecimalFormat formatter = new DecimalFormat("##.##");
                logger.debug(((GasSupplyDemandSim) seller.getSim()).getCurrentDate().toString()  + ": "
+seller.getId() +
                        "  is rescheduling itself to check its AskPrice on :" +
GasSupplyDemandSim.getCalendar().getTime().toString() +
                        ", which should be "+
formatter.format(seller.getTimePeriodDaysToAdjustAskPrice()) + " days in the future"); //$NON-NLS-1$
            }

            this.schedule.scheduleActionAt(nextTime, new ChangeAskPriceAction(seller));

            if (logger.isDebugEnabled()) {
                //double curTime = seller.getSim().getSchedule().getCurrentTime();
                NumberFormat formatter = NumberFormat.getCurrencyInstance();
                //GasSupplyDemandSim.getCalendar().setTimeInMillis((new
Double(curTime)).longValue());
```

```
                                logger.debug("Current time is " +
((GasSupplyDemandSim)seller.getSim()).getCurrentDate().toString() + ", " + seller.getId() + " (oldAskPrice,
newAskPrice ) = ( " + formatter.format( oldAskPrice ) + ", " + formatter.format( newAskPrice ) + " )" );


                        }


                }


}

package com.abcecon.model.sellers;

import java.awt.Color;
import java.awt.Image;
import java.awt.geom.Rectangle2D;
import java.net.URL;
import java.util.ArrayList;
import java.util.Calendar;

import javax.swing.ImageIcon;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;

import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.engine.SimModel;
import uchicago.src.sim.gui.SimGraphics;
import uchicago.src.sim.network.EdgeFactory;

import com.abcecon.controller.GasSupplyDemandSim;
import com.abcecon.model.CheckInventoryAction;
import com.abcecon.model.MarketAgent;
import com.abcecon.model.SIMUTIL;
import com.abcecon.model.TIME;
import com.abcecon.model.Transaction;
import com.abcecon.model.buyers.ISupplierSelector;


public class Seller extends MarketAgent {


        class CheckGasAction extends BasicAction{

                Logger LOGGER = Logger.getLogger(CheckGasAction.class);
```

```
		MarketAgent agent;

		public CheckGasAction(MarketAgent _agent){
			// Set up the Log4j logger
			URL configURL = Loader.getResource(SIMUTIL.LOG4FILE);
			PropertyConfigurator.configure(configURL);

			this.agent = _agent;
		}

		/* (non-Javadoc)
		 * @see uchicago.src.sim.engine.BasicAction#execute()
		 */
		@Override
		public void execute() {

			double curTime = sim.getSchedule().getCurrentTime();

			// check how much gas we have relative to our trigger level to buy
			if( agent.getInventory() < ( agent.getCapacity() *
agent.getFractionOfCapacityTriggerToBuy() )){
				// then I need to buy gas. Schedule it
				sim.getSchedule().scheduleActionAt(curTime + DELTA_TIME_STEP, new
CheckGasAction(agent));

				if( LOGGER.isDebugEnabled()){
					Calendar utilCal = TIME.createUtilityCalendar(curTime +
DELTA_TIME_STEP);
					LOGGER.debug(
((GasSupplyDemandSim)sim).getCurrentDate().toString() +": " + agent.getId() + " needs to buy gas at future time "
+ utilCal.getTime().toString());

				}



				// find a producer
				ISupplierSelector producerSelector = ((Seller) agent).getSupplierSelector();
				if( producerSelector != null){
					MarketAgent producer = producerSelector.findSupplier();

					// how much do I need to buy?
					double amountToFillTank = (agent).getCapacity() -
(agent).getInventory();

					// how much is the producer selling for, at this point we are buying it
from them.
					double pricePaidPerGallon = producer.getAskPrice();

					// create a new Transaction(double _time, MarketAgent _buyer,
MarketAgent _seller, double _amount, double _price){
```

```
                                        Transaction transaction = new
Transaction(sim.getSchedule().getCurrentTime(), agent, producer, amountToFillTank, pricePaidPerGallon );

                                        // ADD edge on graph


        agent.setLastTransactionEdge(EdgeFactory.createDrawableEdge(agent, producer));
                                        // update the display
                                        ((GasSupplyDemandSim)
sim).getSpatialDisplaySurface().updateDisplay();

                                        //add gas to seller's tank
                                        agent.setInventory(agent.getInventory() + amountToFillTank );

                                        // take money out of seller's bank account
                                        agent.setWealth(agent.getWealth() - amountToFillTank *
pricePaidPerGallon);

                                        // add money to producer's bank account
                                        producer.setWealth(producer.getWealth() + amountToFillTank *
pricePaidPerGallon);

                                }else{
                                        LOGGER.fatal("No implementor of IProducerSelector assigned to " +
agent.getId());

                                }

                        }else{
                                // then have them check there gas in a few days
                                sim.getSchedule().scheduleActionAt(curTime +
TIME.daysToSimulationDuration(NEXT_TIME_TO_CHECK_DAYS), new CheckGasAction(agent));

                        }

                }

        }



        /** A static counter to help us create a unique id of each agent. The reason we use "static" is that this
means that this number is the SAME for ALL instances
         * of Agents, that is it resides inside the Class instead of inside each instance of an MarketAgent Class.
         */
        private static int counter;

        private double hoursOfOperationPerDay;


        /**
```

```
	 * Fixed costs are independent of whether the seller is selling product or not. This is the cost of operation per (day) independent of what is sold that day.
	 */
	private double fixedCostsPerDay;

	/**
	 * The total cost of labor per hour of operation. This is not per employee, but total per hour of open operation.
	 */
	private double laborCostsPerHour;

	/**
	 * The store of all buy transactions, Seller buying gas from Producers.
	 */
	private ArrayList<Transaction> buyTransactions;

	/**
	 * The store of all sell transactions, Seller selling gas to GasBuyers.
	 */
	private ArrayList<Transaction> sellTransactions;


	private static Image sellerPicture;

	/**
	 * Strategy pattern for finding a producer. Different Sellers will use different methods
	 */

	private ISupplierSelector producerSelector;

	/**
	 * @param _simulation
	 * @param x
	 * @param y
	 */
	public Seller(SimModel _simulation, double x, double y) {
		super(_simulation, x, y);

		// increment the count and assign the unique id
		super.id = "GasSeller_" + counter; // Java Strings are smart enough now to create a String out of a integer combined with a String.

		loadGasSellerImage();
		this.setNodeLabel(this.id);

		this.buyTransactions = new ArrayList<Transaction>();
		this.sellTransactions = new ArrayList<Transaction>();

		Seller.counter++; // increment the counter. Note how I call the static field here.

	}
```

```
	/**
	 * This method is where we autonomously schedule ourself.  This agent can schedule and reschedule future actions, and change actions on its own.
	 *
	 */
	@Override
	public void initialize() {
		Schedule schedule = this.getSim().getSchedule();

		// CHECK/UPDATE ASK PRICE
		double simTimeAskPrice = TIME.daysToSimulationDuration( this.timePeriodDaysToAdjustAskPrice);
		schedule.scheduleActionAt(simTimeAskPrice, new ChangeAskPriceAction(this));


		// CHECK INVENTORY AND BUY PRODUCT: add the check inventory action to the schedule, it should reschedule itself inside the CheckInventoryAction class.
		double simTimeCheckInv = TIME.daysToSimulationDuration( this.getFrequencyDaysCheckInventory() );
		schedule.scheduleActionAt(simTimeCheckInv, new CheckInventoryAction(this)); // Note that CheckInventoryAction can also BUY product.


	}




	/**
	 * The fixed costs do not depend on how much is produced.
	 *
	 * @return the fixedCostsPerDay
	 */
	public double getFixedCostsPerDay() {
		return fixedCostsPerDay;


	}


	/**
	 *
	 *
	 * @param fixedCostsPerDay the fixedCostsPerDay to set
	 */
	public void setFixedCostsPerDay(double fixedCosts) {
		this.fixedCostsPerDay = fixedCosts;
	}
```

```
/**
 * @return the producerSelector
 */
public ISupplierSelector getSupplierSelector() {
	return producerSelector;
}



/**
 * @param producerSelector the producerSelector to set
 */
public void setProducerSelector(ISupplierSelector producerSelector) {
	this.producerSelector = producerSelector;
}



/**
 * @return the hoursOfOperationPerDay
 */
public double getHoursOfOperationPerDay() {
	return hoursOfOperationPerDay;
}



/**
 * @param hoursOfOperationPerDay the hoursOfOperationPerDay to set
 */
public void setHoursOfOperationPerDay(double hoursOfOperationPerDay) {
	this.hoursOfOperationPerDay = hoursOfOperationPerDay;
}



/**
 * Returns the variable costs of the Seller in units of $/unit produced.
 *
 * @return the laborCostsPerHour
 */
public double getLaborCostsPerHour() {
```

```
		return laborCostsPerHour;
	}


	/**
	 * @param laborCostsPerHour the laborCostsPerHour to set
	 */
	public void setLaborCostsPerHour(double variableCosts) {
		this.laborCostsPerHour = variableCosts;
	}


	/**
	 * @return the sellerPicture
	 */
	public static Image getSellerPicture() {
		return sellerPicture;
	}




	/**
	 * @return the buyTransactions
	 */
	public ArrayList<Transaction> getBuyTransactions() {
		return buyTransactions;
	}




	/**
	 * @return the sellTransactions
	 */
	public ArrayList<Transaction> getSellTransactions() {
		return sellTransactions;
	}



	private static void loadGasSellerImage() {
		if (sellerPicture == null) {
			// would like to change this to have a test to see if the agent is a seller or not, and then
change color accordingly... but it is static.
			java.net.URL sellerPicURL = MarketAgent.class.getResource("gasPump.gif");
			sellerPicture = new ImageIcon(sellerPicURL).getImage();
		}
	}
```

```
	@Override
	public void draw(SimGraphics g) {
		// draw the agent's picture
		g.drawImage(sellerPicture);

		// grab the width of the picture
		int width = sellerPicture.getWidth(null);

		g.setFont(super.getFont());

		// get the size of the node's text
		Rectangle2D bounds = g.getStringBounds(this.getNodeLabel());

		// set the graphics to draw the text above the label
		// the x coordinate is relative to the upper left corner of the image
		// so the coordinates are shifted to account for that
		g.setDrawingCoordinates((float) (this.getX() + width / 2.0 - bounds.getWidth() / 2.0),(float)
(this.getY() - bounds.getHeight() - 2), 0f);

		// draw the label
		g.drawString(getNodeLabel(), Color.BLACK);
	}



	public String[] getProbedProperties() {
		return new String[] {"id", "wealth"};
	}




}
package com.abcecon.view;


import org.apache.log4j.Logger;


import java.text.NumberFormat;

import java.util.List;


import uchicago.src.sim.analysis.OpenSequenceGraph;

import uchicago.src.sim.analysis.Sequence;
```

```
import uchicago.src.sim.engine.BasicAction;

import uchicago.src.sim.engine.SimModel;

import cern.colt.list.DoubleArrayList;

import cern.jet.stat.Descriptive;


import com.abcecon.controller.GasSupplyDemandSim;

import com.abcecon.model.MarketAgent;

import com.abcecon.model.TIME;


public class AverageSellerPriceTimeSeriesGraph {

		/**

		 * Logger for this class

		 */

		private static final Logger logger = Logger.getLogger(AverageSellerPriceTimeSeriesGraph.class);




		public static final double TIMEAXIS_MIN = 0.0;

		public static final double TIMEAXIS_MAX = TIME.weeksToSimulationDuration(8.0);

		public static final double YAXIS_MIN = 1.90;

		public static final double YAXIS_MAX = 2.20;


		protected OpenSequenceGraph graph;



		protected int updateGraphFrequencyDays;
```

```
protected SimModel sim;



public AverageSellerPriceTimeSeriesGraph(SimModel _sim, int _updateGraphFrequencyDays){
    this.sim = _sim;
    this.graph = new OpenSequenceGraph("Average Seller Price", this.sim);
    // set axes up
    this.graph.setAxisTitles("time", "mean seller price");
    this.graph.setXRange(TIMEAXIS_MIN, TIMEAXIS_MAX);
    this.graph.setYRange(YAXIS_MIN, YAXIS_MAX);
    this.graph.setXAutoExpand(true);
    this.graph.setYAutoExpand(true);



    this.updateGraphFrequencyDays = _updateGraphFrequencyDays;

    // add the sequence implementation
    graph.addSequence("mean seller price", new MeanPriceSequenceImpl(this.sim));

    // put ourselves on the Schedule
    sim.getSchedule().scheduleActionAt(sim.getSchedule().getCurrentTime() +
TIME.daysToSimulationDuration(_updateGraphFrequencyDays),
                        new UpdateGraphAction());

    // display ourselves
```

```
        graph.display();
    }


    /**
     * Returns the graph that is wrapped by this class. You need to use this graph instance for scheduling the updates and manipulating
     * the OpenSeqenceGraph.
     *
     * @return the graph
     */
    public OpenSequenceGraph getGraph() {
        return graph;
    }


    class MeanPriceSequenceImpl implements Sequence{

        List<MarketAgent> gasSellers;

        public MeanPriceSequenceImpl(SimModel _sim){
                this.gasSellers = ( (GasSupplyDemandSim) sim).getGasSellers();
```

```
		}
		/* (non-Javadoc)
		 * @see uchicago.src.sim.analysis.Sequence#getSValue()
		 */
		public double getSValue() {

			double meanPrice = 0.0;

			DoubleArrayList salesPrices = new DoubleArrayList(); // using the CERN Stats library here, fast, easy.
			for(int i = 0; i < this.gasSellers.size(); i++){
				MarketAgent seller = this.gasSellers.get(i);
				salesPrices.add(seller.getAskPrice());
			}

			meanPrice = Descriptive.mean(salesPrices);

			if (logger.isDebugEnabled()) {
				NumberFormat formatter = NumberFormat.getCurrencyInstance();
				logger.debug( ( (GasSupplyDemandSim) sim).getCurrentDate().toString() + ": mean sales price = " + formatter.format(meanPrice)); //$NON-NLS-1$
			}

			return meanPrice;
		}

	}
```

```
	/**
	 * We need a BasicAction to schedule ourselves for updating and displaying, instead of doing it every simulation step.
	 *
	 * @author mackerrow  Mar 9, 2009   4:45:36 AM
	 *
	 */
	class UpdateGraphAction extends BasicAction{



	    public UpdateGraphAction(){
	    }


	    /* (non-Javadoc)
	     * @see uchicago.src.sim.engine.BasicAction#execute()
	     */
	    @Override
	    public void execute() {
	        // this should update the data calculation in the graph, and then redisplay things.
	        graph.record();
	        graph.updateGraph();


	        // reschedule
	        sim.getSchedule().scheduleActionAt( sim.getSchedule().getCurrentTime() + TIME.daysToSimulationDuration(updateGraphFrequencyDays),
	                                            new UpdateGraphAction());
	    }
```

```
		}


}
package com.abcecon.view;

import uchicago.src.sim.analysis.Plot;
import uchicago.src.sim.engine.SimModel;

public class SupplyDemandPlot {

		private Plot plot;


		SimModel sim;

		public SupplyDemandPlot(SimModel _sim){
		    this.sim = _sim;
		}

}
```