Writing a Discrete Event Simulation: ten easy lessons

The purpose of this page is to address the frequently asked question "How do I write a discrete event simulation?" Although there are a number of good software libraries for simulation, including one that I helped write, this page will show you that the a basic simulation program can be put together without too much effort. Nonetheless, many of the available libraries provide features that you would not want to have to recreate yourself and you should look at these libraries for any large projects By working through the lessons you will understand what goes into a simulation package and will hopefully be in a better position to either write your own simulations or evaluate existing software libraries. You are also encouraged to look at the simulation sections in my bookmarks page and the comp.simulation software archive.

The following pieces are essential to any simulation

- An abstract framework of events
- A data structure to manage events
- Functions to generate random variates
- Facilities to allows objects to interact

I strongly believe that object-oriented programing should be used for discrete-event simulations. In a pinch you can write an object oriented program in any language, but I strongly recommend a language that supports it well, like SIMULA, Smalltalk, object-pascal, objective-C, Python, C++, or Java. There are a few languages that have builtin support for mutiple threads of execution, as in SIMULA, Smalltalk and Java. At one time I wrote models using a concurrent extension to C++ that provided thread support (C++SIM), but dropped that in favor of the simpler and more universal stategy used in these lessons.

I have written the examples for these lessons in Java. If you forget about all the fanfare about Java as a language for web pages, it is also a very good, clean object oriented language that is safer and easier (most of the time) to use than C++. I hope that these lessons will be useful even for folks who want to write simulation in a language other than Java. Much of the syntax would be self explanatory for anyone familiar with another object-oriented language, but here are a couple of tips for those of you new to Java. There also also many good introductory web pages and books about java listed at ???. For simplicity I have put each lesson in a single file, but you can also obtain them separately. The single may cause your java compiler to throw warnings, so you can put each class in a separate file if you want. You could also put these classes in a separate package if you are the neat and tidy type. You can also obtain the contents of this entire directory as des.zip .

Java Hints

- There is no explicit memory deallocation; unreferenced objects are garbage collected by the runtime system.
- Class variables are marked as `static` and are initialized in a block of code that is labeled `static` .
- There is no enumerated type, but constants can be defined as `final static int`.
- Class methods are marked `static` and can serve the role of global functions that are not attached to particular objects.
- An interface is an abstract class. An interface is only allowed to defined the signature of methods. It cannot define attribute variables or implement any methods.
- The counterpart to a C or C++ `main` function is a static member of any class with the signature `public static void main(String[] args)` .

Lesson 1: The Bare-bones Abstraction

Abstractly, a discrete event simuation consists of a bunch of events and a central simulator object that executes these

events in order. In a procedural language, "executing an event" boils down to calling a function. Some languages support a very flexible notion for representing a function call that you want to invoke later, called closures. Neither C++ or java support this at present, so we take the simpler approach that executing an event means calling the execute method on a particular event object.

To make the code more flexible, we describe the system abstractly in terms of one interface and three classes. The `Comparable` interface defines the `lessThan()` method that is used by the `OrderedSet` interface to sort the events. In this first lesson no assumptions are made about how the order among events is defined. The implementation of the `OrderedSet` is left for lesson 3.

Copy the following code to a file called lesson1.java. Then compile it by executing `javac lesson1.java` . The source is also available as a separate file .

```
interface Comparable {
    boolean lessThan(Comparable y);
}

abstract class  AbstractEvent implements Comparable {
    abstract void execute(AbstractSimulator simulator);
}

abstract class OrderedSet {
    abstract void insert(Comparable x);
    abstract Comparable  removeFirst();
    abstract int size();
    abstract Comparable remove(Comparable x);
}

class AbstractSimulator {
    OrderedSet events;
    void insert(AbstractEvent e) {
        events.insert(e);
    }
    AbstractEvent cancel(AbstractEvent e)  {
        throw new java.lang.RuntimeException("Method not implemented");
    }
}
```

Lesson 2: Order Events By Time

Although some situations might require events to be ordered by priority or by their order of creation, all of the examples below treat events as ordered by time. This allows us to create concrete classes that implement `Event` and flesh out the `Simulator` class. Note: Separating `AbstractSimulator` from `Simulator` is a good idea from the design point of view, but it complicates some of the examples below. If you prefer, you can merge the two classes and commit yourself to simulations in which events are ordered by time. The source is also available as a separate file .

```
abstract class Event extends AbstractEvent {
    double time;
    public boolean lessThan(Comparable y) {
        Event e = (Event) y;  // Will throw an exception if y is not an Event
        return this.time < e.time;
    }
}

class Simulator extends AbstractSimulator {
    double time;
    double now() {
        return time;
    }
```

```
    void doAllEvents() {
        Event e;
        while ( (e= (Event) events.removeFirst()) != null) {
            time = e.time;
            e.execute(this);
        }
    }
}
```

Lesson 3: Implement OrderedSet

From the technical point of view, developing a good implementation of the event set is one of the most challenging parts of building a simulation. The very minimal interface defined above could be implemented using a binary tree that is kept in heap order. Most introductory data structure books cover heaps. They are not too hard to implement and provide O(log n) performance, where n is the number of events that are waiting. However, in some situations a model also needs to be able to remove events. For example, if a person dies, any future events for that person probably need to be immediately cancelled. Supporting removal in O(log n) time requires that the binary tree be kept completely sorted. The well known problem with sorted trees is that one must be careful to keep them fairly balanced so that performance does not degrade. There are many ways to do this, and there are even some data structures that achive O(1) performance when properly tuned. For large simulations, a serious modeler should look at some of the algorithms collected by Douglas Jones , look at the code that is part of SimPack and SIMEX, or use the author's splay tree classes. Below is a class that uses the `java.util.Vector` class and has performance O(n) because it does a linear search when it inserts an item. I have added a remove method such as one might need. The source is also available as a separate file .

```
class ListQueue extends OrderedSet {
java.util.Vector elements = new java.util.Vector();
void insert(Comparable x) {
int i = 0;
while (i < elements.size() && ((Comparable) elements.elementAt(i)).lessThan(x)) {
i++;
}
elements.insertElementAt(x,i);
}
Comparable removeFirst() {
if (elements.size() ==0) return null;
Comparable x = (Comparable) elements.firstElement();
elements.removeElementAt(0);
return x;
}
Comparable remove(Comparable x) {
for (int i = 0; i < elements.size(); i++) {
if (elements.elementAt(i).equals(x)) {
Object y = elements.elementAt(i);
elements.removeElementAt(i);
return (Comparable) y;
}
}
return null;
}
public int size() {
return elements.size();
}

}
```

Lesson 4: A Trivial Example

We now have enough of a code framework to build our first simulation. To build a model we must construct the various types of events and define their behavior, and extend the `Simulator` class so that the main program creates the proper initial conditions and then runs the simulation. For this example, the events will simply print out the current time. The simulation will be initialized by creating five events that are scheduled to go off at various times.

After compiling this lesson and all of the previous lessons, you can run this example by typing `java BeepSimulator` . The source is also available as a separate file .

```
class Beep extends  Event {
    Beep(double time) {
        this.time = time;
    }
    void execute(AbstractSimulator simulator) {
        System.out.println("The time is "+time);
    }
}

class BeepSimulator extends Simulator {
    public static void main(String[] args) {
        new BeepSimulator().start();
    }
    void start() {
        events = new ListQueue();
        insert( new Beep(4.0));
        insert( new Beep(1.0));
        insert( new Beep(1.5));
        insert( new Beep(2.0));

        doAllEvents();
    }
}
```

Lesson 5: Events that Reschedule More Events

The model in lesson 4 was pretty boring because the simulation entailed nothing more than executing the events that were created initially. In this lesson, we show how an event can schedule other events, or reschedule itself. This illustrates the rationale for passing the simulator as an argument to the execute method. In order for an event to schedule events, it needs to call the BaseSimulator.insert method on the current simulator object. This simulation creates an event that reschedules itself every 2.0 time units, up to time 10.0. The source is also available as a separate file .

```
class Counter extends Event {
    Counter(double time) {
        this.time = time;
    }
    void execute(AbstractSimulator simulator) {
        System.out.println("The time is "+time);
        if (time <10) {
            time += 2.0;
            simulator.insert(this);
        }
    }
}
class class CounterSimulator extends Simulator {
    public static void main(String[] args) {
        new CounterSimulator().start();
    }
```

```
    void start() {
        events = new ListQueue();
        insert( new Counter(0.0));
        doAllEvents();
    }
}
```

Lesson 6: Randomness

So far the examples have been deterministic; there has been no random component. In most realistic discrete event simulations the creation and timing of events is stochastic. To support this we need to be able to generate random variates that are drawn from a variety of distributions. Algorithms to do this are covered in most books on simulations as well as the standard reference by DeVroye. Code for random variates is also in most of the packages listed in the software bibliography. For the time being we will only implement the exponential distribution and leave it to you to implement others as they are needed. This lesson's model creates five radioactive particles with random lifespans. The source is also available as a separate file .

```
class Random {
    static double exponential(double mean) {
        return - mean * Math.log(Math.random());
    }
  static boolean bernoulli(double p) {
    return Math.random() < p;
  }
    /* .. and other distributions */
}

class RadioactiveParticle extends Event {
    RadioactiveParticle(double time) {
        this.time = time;
    }
    void execute(AbstractSimulator simulator) {
        System.out.println("A particle disintigrated at t = "+time);
    }
}

class ParticleSimulator extends Simulator {
    public static void main(String[] args) {
        new ParticleSimulator().start();
    }
    void start() {
        events = new ListQueue();
        /* Create 5 particle, each with a mean lifespan of 4 time units */
        insert( new RadioactiveParticle(Random.exponential(4.0)) );
        insert( new RadioactiveParticle(Random.exponential(4.0)) );
        insert( new RadioactiveParticle(Random.exponential(4.0)) );
        insert( new RadioactiveParticle(Random.exponential(4.0)) );
        insert( new RadioactiveParticle(Random.exponential(4.0)) );

        doAllEvents();
    }
}
```

Lesson 7: Interactions, part 1: A Queue

The final type of complexity that one finds in discrete event models is the ability for objects in the simulation to interact. This is the feature that really makes discrete event systems unpredictable and worth simulating. In manufacturing and business applications the most common way in which objects interact is via queues. Objects compete for resources and some sort of queuing protocol determines the order of access to resources. This lessons simulates the classical M/M/1 queuing model, in which customers arrive according to a Poisson process (a Markov process) and are served by a single server for an exponentially distributed amount of time (service is a Markov process). Customers are generated for 8 hours according to a Poisson process. These customers wait in a first-in-first-out queue for a sever with an exponentially distributed service time. This model also shows that objects that interact need some way to know about each other. For this model, the `Generator` is connected to the `Queue` so that new customers can be added to the `Queue`. The `Queue` is connected to `Server` so that arriving customers can proceed directly to the `Server` if it is available. The `Server` is connected to the `Queue` so it can get another customer after it finishes servicing one. The source is also available as a separate file .

```
/**
* This class is empty because customers are just passive tokens that
* are passed around the system.
*/
class Customer {
}
/**
* Generate a stream of customers for 8.0 time units.
*/
class Generator extends Event {
    Queue queue;
    /**
    * Create a new Customer.  Add the customer to the queue  and
    * schedule the creation of the next customer
    */
    void execute(AbstractSimulator simulator) {
        Customer customer = new Customer();
        queue.insert(simulator, customer);
        time += Random.exponential(8.0);
        if (time < 10.0) simulator.insert(this);
    }
}

class Queue {
    /**
    * Use the Java Vector to implement a FIFO queue.
    */
    private java.util.Vector customers = new java.util.Vector();
    Server server;
    /**
    * Add a customer to the queue.
    * If the server is available (whic also implies this queue is empty),
    * pass the customer on to the server.
    * Otherwise add the customer to the queue.
    */
    void insert(AbstractSimulator simulator, Customer customer) {
        if (server.isAvailable()) {
            server.insert(simulator,customer);
        } else {
            customers.addElement(customer);
        }
    }
    /**
    * @return the first customer in the queue
    */
```

```
    Customer remove() {
        Customer customer = (Customer) customers.firstElement();
        customers.removeElementAt(0);
        return customer;
    }
    int size() {
        return customers.size();
    }
}

/**
* A server that holds a customer for an exponentially distributed amout of time
* and releases it.
*/
class Server extends Event {
    private Customer customerBeingServed;
    Queue queue;
    /**
    * The customer's service is completed so print a message.
    * If the queue is not empty, get the next customer.
    */
    void execute(AbstractSimulator simulator) {
        System.out.println("Finished serving " + customerBeingServed + " at time " + time);
        customerBeingServed = null;
        if (queue.size()>0) {
            Customer customer = queue.remove();
            insert((Simulator) simulator, customer);
        }
    }
    boolean isAvailable() {
        return (customerBeingServed == null);
    }
    /**
    * Start a customer's service.  The simulator must be passed in
    * as a parameter so that this can  schedule the time
    * when this server will be done with the customer.
    */
    void insert(AbstractSimulator simulator, Customer customer) {
        if (customerBeingServed != null) {
            /* Should never reach here */
            System.out.println("Error: I am busy serving someone else");
            return;
        }
        customerBeingServed = customer;
        double serviceTime = Random.exponential(1.0);
        time = ((Simulator)simulator).now() + serviceTime;
        simulator.insert(this);
    }
}

class BankSimulator extends Simulator {
    public static void main(String[] args) {
        new BankSimulator().start();
    }
    void start() {
        events = new ListQueue();

        /* Create the generator, queue, and simulator */
        Generator generator = new Generator();
        Queue queue = new Queue();
        Server server = new Server();

        /* Connect them together. */
```

```
        generator.queue = queue;
        queue.server = server;
        server.queue = queue;

        /* Start the generator by creating one customer immediately */
        generator.time = 0.0;
        insert(generator);

        doAllEvents();
    }
}
```

Lesson 8: Interactions, part 2: A Mixing Structure

Biological models rarely use queues. Instead, interactions are mediated by social groups, physical proximity and communication channels. This lesson implements the clasical Susceptible-Infectious-Recovered model for the spread of an infectious disease through a closed population. It might be taken as a rough approximation to the spread of a new strain of Influenza through a school. In this model, we enable the people to find each other for interacting by creating a "global" variable `population` that stores the set of all people. During a person infectious period, s/he makes contacts with randomly selected members of the population. This set is maintained by inserting newly created Person objects into the set. If the population underwent fluctuations due to births, deaths, or migration, the population would have to be notified of each change. If these population changes need to be handled efficiently, the Vector data structure used in this lesson is not the best. You should look at using weighted splay trees.

This model shows one way to structure event code in a language where you do not have closures. A Person needs to schedule to two kinds of events, contacts and recovery. This is implemented by having a variable called nextState which tells the execute method which action it needs to perform. This variable is set when the event is scheduled because the code looks ahead to see of another contact will occur before recovery. This code can be generalized to provide a generic finite state machine.

Another way the code could have been structured would have been to create two Event classes. Each event would have a pointer to the Person on which it would act. Then you would also need a way for the recovery event to cancel the contact event that would be scheduled for some time in the future. This approach is covered in lesson 9.

This model also introduces the notion of a model parameter. The average length of the infectious period and the average time between effective contacts are global parameters that are set in the main simulation. This is good coding style because it makes it easier change the model's parameters in a central location. The source is also available as a separate file .

```
class Person extends Event {
    final static int SUSCEPTIBLE = 0;
    final static int INFECTIOUS = 1;
    final static int RECOVERED = 2;
    final static int nStates = 3;
    static double meanInfectiousDuration;
    static double meanInterContactInterval;

    /**
     *  A vector with every person in the population
    */
    static java.util.Vector population;
    /**
     * A count of the number of people in each state
     */
    static int count[];
    static {
        population = new java.util.Vector();
        count = new int[nStates];
```

```
        for (int i=0; i< nStates; i++) {
          count[i] = 0;
        }
    }

    int state;
    int nextState;
    double recoveryTime;

    Person (int initialState) {
        state = initialState;
        population.addElement(this);
        count[state]++;
    }
    void execute(AbstractSimulator simulator) {
        if (nextState == INFECTIOUS) { // action is make contact
            makeContact((Simulator) simulator);
            scheduleNextEvent((Simulator) simulator);
        } // else action is recover
        count[state]--;
        state = nextState;
        count[state]++;
        display((Simulator) simulator);
    }

    /**
    * Display the current population distribution
    */
    void display(Simulator simulator) {
        System.out.print("t= " + simulator.now());
        for (int i=0; i< nStates; i++) {
          System.out.print(" count[" + i + "]= " + count[i]);
        }
        System.out.println("");
    }

    /**
    * Pick a random member of the population and infect the person if it
    * is susceptible.
    */
    void makeContact(Simulator simulator) {
        Person person = (Person) population.elementAt(
          (int) (population.size() * Math.random()));
        if (person.state != SUSCEPTIBLE) return;
        person.recoveryTime = simulator.now() + Random.exponential(meanInfectiousDuration);
        person.scheduleNextEvent(simulator);
    }
    /**
    * Schedule the next event, which will be a contact or recovery, whichever
    * comes first.
    */
    void scheduleNextEvent(Simulator simulator) {
        double nextContactTime = simulator.now() + Random.exponential(meanInterContactInterval);
        if (nextContactTime < recoveryTime) {
            nextState = INFECTIOUS;
            time = nextContactTime;
        } else {
            nextState = RECOVERED;
            time = recoveryTime;
        }
        simulator.insert(this);
    }
}
```

```
class DiseaseSimulator extends Simulator {
    public static void main(String[] args) {
        new DiseaseSimulator().start();
    }
    void start() {
        events = new ListQueue();

        Person.meanInfectiousDuration = 10.0;
        Person.meanInterContactInterval = 4.0;
        Person person;
        for (int i = 0; i< 995; i++) {
            new Person(Person.SUSCEPTIBLE);
        }
        for (int i = 0; i< 5; i++) {
            person = new Person(Person.INFECTIOUS);
            person.recoveryTime = Random.exponential(Person.meanInfectiousDuration);
            person.scheduleNextEvent(this);
        }

        doAllEvents();
    }
}
```

Lesson 9: Mutiple Activities

Models of human behavior often need a single person to be involved in multiple activities. The model in lesson 8 provides a taste of this; an infectious person needs to schedule both contacts and recovery. This lesson develops a more elaborate model for the spread of a disease through a population. This model is adds some of the features that are needed for a sexually transmitted disease. Now individuals can have a partner (but at most one) with whom they make most of their contacts. The model takes into account partnership formation and dissolution.

To allow a person to respond to the different kinds of events that can occur, we create an abstract interface called `MessageHandler` . When an event goes off it passes a string that describes the type of event that occured to the person via the `handle(message)` method.

This implementation also introduces the idea of canceling events. In previous lessons, all of the events that were schedules were sure to happen. Sometimes it is easier to write models in such a way that an event may be tentatively scheduled and may be canceled due to events that occur execution time. In this model, when person A selected person B as a partner, person B needs to cancel its event when it planned to select a partner. Also, whe a person recovers from the disease, it needs to cancel the next effective contact that it had planned to make. In lesson 8 we avoided this by testing whether recovery would occur before the next contact, but this lesson shows a different implementation strategy. The source is also available as a separate file . There are a few things that are set up in this lesson with extra flexibility so that lesson 10 can re-use most of this code without modification.

```
interface MessageHandler {
  void handle(Message message);
}


/**
 * An event that, when executed, passes a message to
 * a particular MessageHandler.
*/
class Message extends Event {
  MessageHandler messageHandler;
  String command;
```

```
  AbstractSimulator simulator;
  Message(MessageHandler messageHandler, String command) {
    this.messageHandler = messageHandler;
    this.command = command;
  }
  Message(MessageHandler messageHandler) {
    this.messageHandler = messageHandler;
    this.command = null;
  }
  void set(String command, double time) {
    this.command = command;
    this.time = time;
  }
  void execute(AbstractSimulator simulator) {
    this.simulator = simulator;
    if (command != null && messageHandler != null) {
      messageHandler.handle(this);
    }
  }
}

/**
 * A person who may have a partner.
 * Regardless of whether there is a current partner, the person
 * makes effective contacts at a constant rate.
 * If there is no current partner, then the contact is made with
 * a random member of the population.  If there is a partner,
 * the partner or a random member of the population.
 */
class SexualPerson implements MessageHandler {
  /**
   * This is a secondary messageHandler that we pass messages
   * to after we have handled the meessage.  The idea is that the
   * SexualDisease class will handle the process as it relates to
   * changing the state of individuals, while the display object
   * will handle the displaying of the state
   */
  static MessageHandler display;
  SexualPerson partner = null;
  Message partnershipMessage = new Message(this);
  Message diseaseMessage = new Message(this);
  Message contactMessage = new Message(this);
  int diseaseState;
  final static int SUSCEPTIBLE = 0;
  final static int INFECTIOUS = 1;
  final static int RECOVERED = 2;
  static java.util.Vector population;
  static int countSusceptible = 0;
  static int countInfectious = 0;
  static int countRecovered = 0;
  static int countPartnerships = 0;

  static double meanInfectiousDuration;
  static double meanPartnershipDuration;
  static double meanInterContactTime;
  static double meanInterPartnershipTime;
  static double probabilityOfExtrapartnershipContact;
  static {
    meanInfectiousDuration = 2.0;
    meanPartnershipDuration = 0.5;
    meanInterContactTime = 1.0;
    meanInterPartnershipTime = 0.3;
    probabilityOfExtrapartnershipContact = 0.1;
```

```
  clearPopulation();
}

static public void clearPopulation() {
  population = new java.util.Vector();
  countSusceptible = 0;
  countInfectious = 0;
  countRecovered = 0;
  countPartnerships = 0;
}
/**
 * return a random infectious period
 */
double infectiousDuration() {
  return Random.exponential(meanInfectiousDuration);
}
/**
 * Return a random time between effective contacts
 */
double interContactTime() {
  return Random.exponential(meanInterContactTime);
}
/**
 * Return a random partnership duration
 */
double partnershipDuration() {
  return Random.exponential(meanPartnershipDuration);
}
/**
 * Return a random time between the end of a partnership and
 * the selection of the next partner.
 */
double interPartnershipTime() {
  return Random.exponential(meanInterPartnershipTime);
}
/**
 * Return a random member of the population
 */
static SexualPerson selectFromPopulation () {
  return (SexualPerson) population.elementAt
    ( (int) (population.size() * Math.random()));

}
void incrementCount() {
  switch (diseaseState) {
  case SUSCEPTIBLE:
    countSusceptible ++;
    break;
  case INFECTIOUS:
    countInfectious++;
    break;
  case RECOVERED:
    countRecovered++;
    break;
  }
}
void decrementCount() {
  switch (diseaseState) {
  case SUSCEPTIBLE:
    countSusceptible--;
    break;
  case INFECTIOUS:
    countInfectious--;
```

```
    break;
  case RECOVERED:
    countRecovered--;
    break;
  }
}
void changeDiseaseState(int diseaseState) {
  decrementCount();
  this.diseaseState = diseaseState;
  incrementCount();
}

 SexualPerson(int diseaseState) {
  this.diseaseState = diseaseState;
  incrementCount();
  population.addElement(this);
}
SexualPerson() {
  incrementCount();
  population.addElement(this);
}
/**
 * A callback for the event
 */
public void handle(Message message) {
  if (message.command.equals("recover")) {
    recover((Simulator) message.simulator);
  } else if (message.command.equals("contact")) {
    contact((Simulator) message.simulator);
  } else if (message.command.equals("endPartnership")) {
    endPartnership((Simulator) message.simulator);
  } else if (message.command.equals("beginPartnership")) {
    beginPartnership((Simulator) message.simulator);
  } else if (message.command.equals("recover")) {
    recover((Simulator) message.simulator);
  } else {
    System.out.println("Unknown command: "+ message.command);
  }

  /* Chain the message to the displayer if it exists */
  if (display != null) display.handle(message);
}

/**
 * Print out the time, the number of people in each state
 * and the number of partnerships.
 */
static void printSummary(Simulator simulator) {
  System.out.println("" + simulator.now()+ " " + countSusceptible + " " +
      countInfectious + " " + countRecovered + " " + countPartnerships);
}

/**
 * Make this person infectious
 */
void infect(Simulator simulator) {
  changeDiseaseState(INFECTIOUS);
  diseaseMessage.set(
        "recover",
        simulator.now() + infectiousDuration()
        );
  simulator.insert(diseaseMessage);
  contactMessage.command = "contact";
```

```
  contactMessage.time = simulator.now() + interContactTime();
  simulator.insert(contactMessage);
}

/**
 * Make this person have an effective contact
 */
void contact(Simulator simulator) {
  SexualPerson contactee = null;
  if (partner == null) {
    contactee = selectFromPopulation();
  } else {
    if (Random.bernoulli(probabilityOfExtrapartnershipContact)) {
      contactee = selectFromPopulation();
    } else {
      contactee = partner;
    }
  }
  if (contactee.diseaseState == SUSCEPTIBLE) {
    contactee.infect(simulator);
  }
}

/**
 * Make this person well
 */
void recover(Simulator simulator) {
  simulator.cancel(contactMessage);
  changeDiseaseState(RECOVERED);
}


/**
 * Select a new partner
 */
void beginPartnership(Simulator simulator) {
  SexualPerson person;
  do { // find an unattached person
    person = selectFromPopulation();
  } while (person.partner != null || person == this);
  partner = person;
  partner.beginPartnershipWith(this, simulator);
  partnershipMessage.set (
          "endPartnership",
          simulator.now() + partnershipDuration()
           );
  simulator.insert(partnershipMessage);
  countPartnerships ++;
}

/**
 * This is called to notify this person that
 * it has been selected to be the partner of another person.
 * Set partner and cancel any pending plans to form initiate a partnership
 */
void beginPartnershipWith(SexualPerson person, Simulator simulator) {
  partner = person;
  simulator.cancel(partnershipMessage);
}

/**
 * End this person's current partnership
 */
```

```
  void endPartnership(Simulator simulator) {
    partner.endPartnershipWith(this,simulator);
    partner= null;
    partnershipMessage.set(
                           "beginPartnership",
                           simulator.now() + interPartnershipTime()
                           );
    simulator.insert(partnershipMessage);
    countPartnerships --;
  }
  /**
   * This person's partner decided to end the current partnership
   */
  void endPartnershipWith(SexualPerson person, Simulator simulator) {
    partner= null;
    partnershipMessage.set(
                           "beginPartnership",
                           simulator.now() + interPartnershipTime()
                           );
    simulator.insert(partnershipMessage);
  }
}

/**
  * A helper class that dumps the state of the simulation to System.out
  * this is put is a separate class so that in the next lesson we can replace
  * it with something to handle plotting
  */
class PrintDisplay implements MessageHandler {
  public void handle(Message message) {
    SexualPerson.printSummary((Simulator) message.simulator);
  }
}


class SexualDiseaseSimulator extends Simulator {
  public static void main(String[] args) {
    SexualPerson.display = new PrintDisplay();
    SexualDiseaseSimulator sim = new SexualDiseaseSimulator
        (new ListQueue(),995,5,0,50);
    sim.doAllEvents();
  }
  AbstractEvent cancel(AbstractEvent e) {
    return (AbstractEvent)  events.remove(e);
  }
  SexualDiseaseSimulator(OrderedSet events,
        int initialSusceptible,
        int initialInfectious,
        int initialRecovered,
        int initialPartnerships) {
    this.events = events;
    SexualPerson.clearPopulation();
    SexualPerson person, partner;
    for (int i = 0; i< initialSusceptible; i++) {
      new SexualPerson(SexualPerson.SUSCEPTIBLE);
    }
    for (int i = 0; i< initialInfectious; i++) {
      person = new SexualPerson(SexualPerson.INFECTIOUS);
      person.infect(this);
    }
    for (int i = 0; i< initialPartnerships; i++) {
      person  =  SexualPerson.selectFromPopulation();
      if (person.partner == null) {
```

```
        person.beginPartnership(this);
      }
    }
  }
}
```

Lesson 10: Wrapping a simulation in an Applet

Java has become popular because of its deployability inside browsers, so this lesson shows a simple example of how one might present the model that was developed in lesson 9 as an applet that coulb be viewed in a java-enabled browser.
This codes is available separately as SexualDiseaseSimulatorAppplet.java You would present it to the user with a page like SexualDiseaseSimulatorApplet.html . To provide a nice display I have used a plotting package called ptplot, which is available from http://ptolemy.eecs.berkeley.edu/java/ptplot/ . I have also showed how to used the freely available OrderedSet class that is part of JGL from ObjectSpace to get a faster implementation of the event queue.

```
import java.applet.*;
import java.awt.*;
import ptplot.*;

/**
 * A helper class to let us use JGL OrderedSet for events
 */
class Comparer implements COM.objectspace.jgl.BinaryPredicate {
  public boolean execute(Object x, Object y) {
    return ((Comparable) x) .lessThan((Comparable)y);
  }
}

/**
 * Implement OrderedSet as we define it using a JGL OrderedSet
 */
class JGLOrderedSet extends OrderedSet {
  private COM.objectspace.jgl.OrderedSet set
    = new COM.objectspace.jgl.OrderedSet( new Comparer());
  void insert(Comparable x) {
    set.add(x);
  }
  Comparable  removeFirst() {
    return (Comparable) set.remove(set.begin());
  }
  int size() {
    return set.size();
  }
  Comparable remove(Comparable x) {
    int count = set.remove(x);
    if (count == 0) return null;
    else return x;
  }
}


/**
  * A quick implementation of a tabbed panel to hold
  * component with a CardLayout.  We use this to display
  * the plots.
  */
class TabPanel extends Panel {
```

```
  Button nextButton = new Button("next");
  Button previousButton = new Button("previous");
  Label currentTab = new Label();
  Panel panes = new Panel();
  int current = 0;
  int size = 0;

  TabPanel() {
    panes.setLayout(new CardLayout());
    setLayout( new BorderLayout());

    add("Center", panes);
    Panel buttons = new Panel();
    previousButton.enable(false); // Use setEnabled(false) in JDK 1.1
    buttons.setLayout (new FlowLayout());
    buttons.add(nextButton);
    buttons.add(previousButton);
    buttons.add(currentTab);
    add("South",buttons);
  }
  public Component addTab(String s,Component c) {
    size ++;
    CardLayout cl = (CardLayout) panes.getLayout();
    cl.first(panes);
    currentTab.setText("" + (current+1) + ":" +
size);
    return panes.add(s,c);
  }
  public boolean action(java.awt.Event e, Object arg) {
    CardLayout cl = (CardLayout) panes.getLayout();
    if (e.target == nextButton) {
      cl.next(panes);
      current++;
    }
    else if (e.target == previousButton) {
      cl.previous(panes);
      current --;
    }
    if (current == size-1) nextButton.setEnabled(false);
    else nextButton.setEnabled(true);
    if (current == 0) previousButton.setEnabled(false);
    else  previousButton.setEnabled(true);
    currentTab.setText("" + (current+1) + ":" +
size);
    return super.action(e,arg);
  }
}

/**
  * A convenience class for displaying a label and a text field.
  */
public class LabeledField extends Panel {
  Label label;
  TextField field;
  public  LabeledField(String labelText, double d, int size) {
    this(labelText,String.valueOf(d),size);
  }
  public  LabeledField(String labelText, String defaultValue, int size) {
    setLayout( new BorderLayout());
    label = new Label(labelText);
    add("West",label);
    field = new TextField(defaultValue,size);
    add("East",field);
```

```
  }
  public String getText() {
    return field.getText();
  }
  public void  setText(String s) {
    field.setText(s);
  }
  public void setEditable(boolean b) {
    field.setEditable(b);
  }
  public String asString() {
    return field.getText();
  }
  public double asDouble() {
    String s = field.getText();
    double x  = new Double(s).doubleValue();
    return x;
  }
  public int asInt() {
    String s = field.getText();
    int x  = new Integer(s).intValue();
    return x;
  }
}
/**
  * An applet wrapper around a disease simulation model.
  *
  */
public class SexualDiseaseSimulatorApplet
  extends Applet implements MessageHandler
{
  private double displayInterval = 0.1;
  public static boolean DEBUG = false;
  LabeledField  initialSusceptible = new LabeledField("S(0)","995",4);
  LabeledField  initialInfectious  = new LabeledField("I(0)","5",4);
  LabeledField  initialRecovered = new LabeledField("R(0)","0",4);
  LabeledField  initialPartnerships = new LabeledField("P(0)","50",4);

  LabeledField  meanInfectiousDuration = new LabeledField("infectDur",2.0,4);
  LabeledField  meanPartnershipDuration = new LabeledField("part'pDur",0.5,4);
  LabeledField  meanInterContactTime = new LabeledField("contactRate",0.4,4);
  LabeledField  meanInterPartnershipTime = new LabeledField("part'pRate",0.3,4);
  LabeledField  probabilityOfExtrapartnershipContact
= new LabeledField("probRandomContact",0.1,4);

  Button startButton = new Button("Start");
  Button pauseButton = new Button("Pause");
  Button stopButton = new Button("Stop");
  SexualDiseaseSimulator sim = null;
  Plot plot1 = new Plot();
  Plot plot2 = new Plot();
  Plot plot3 = new Plot();
  boolean firstPoint = true;
  /**
    * Perform one-time initialization
    * Create the gui
    */
  public void init() {
    plot1.setNumSets(1);
    plot1.setXLabel("t"); plot1.setYLabel("Suscectible");
    plot2.setNumSets(1);
    plot2.setXLabel("t"); plot2.setYLabel("Infectious");
    plot3.setNumSets(1);
```

```
    plot3.setXLabel("t"); plot3.setYLabel("Partnerships");
    setLayout( new BorderLayout());

    Panel inputs = new Panel();
    inputs.setLayout ( new GridLayout(0,1));
    inputs.add(initialSusceptible);
    inputs.add(initialInfectious);
    inputs.add(initialRecovered);
    inputs.add(initialPartnerships);
    inputs.add(meanInfectiousDuration);
    inputs.add(meanPartnershipDuration);
    inputs.add(meanInterContactTime);
    inputs.add(meanInterPartnershipTime);
    inputs.add(probabilityOfExtrapartnershipContact);
    add("West",inputs);

    Panel buttons = new Panel();
    buttons.setLayout(new FlowLayout());
    buttons.add(startButton);
    buttons.add(pauseButton);
    buttons.add(stopButton);
    add("North",buttons);
    TabPanel plots = new TabPanel();
    plots.addTab("t vs S",plot1);
    plots.addTab("t vs I",plot2);
    plots.addTab("t vs P",plot3);
    add("Center",plots);
    pauseButton.setEnabled(false);
    stopButton.setEnabled(false);
    super.init();
    plot1.init();
    plot2.init();
    plot3.init();
  }

  public void startSimulation() {
    SexualPerson.display = this;
    SexualPerson.meanInfectiousDuration = meanInfectiousDuration.asDouble();
    SexualPerson.meanPartnershipDuration = meanPartnershipDuration.asDouble();
    SexualPerson.meanInterContactTime = meanInterContactTime.asDouble();
    SexualPerson.meanInterPartnershipTime =meanInterPartnershipTime.asDouble()
;
    SexualPerson.probabilityOfExtrapartnershipContact
      = probabilityOfExtrapartnershipContact.asDouble();

    sim = new SexualDiseaseSimulator(new JGLOrderedSet(),
         initialSusceptible.asInt(),
         initialInfectious.asInt(),
         initialRecovered.asInt(),
         initialPartnerships.asInt());
    stopButton.setEnabled(true);
    pauseButton.setEnabled(true);
    pauseButton.setLabel("Pause");
    firstPoint = true;
    Message displayMessage = new Message(this,"display");
    displayMessage.time = 0.0;
    sim.insert(displayMessage);
    runSimulation();
  }

  /**
    * Start the simulation running in a separate thread.
    */
```

```
public void runSimulation() {
  sim.state = Simulator.RUNNING;
  new Thread () {
    public void run() {
      sim.doAllEvents();
    }
  }.start();
}

/**
  * Callback for the Stop button
  *Stop and disable the current simulation
  */
public void stopSimulation() {
  sim.state = Simulator.PAUSED;
  sim.events = null;
  stopButton.setEnabled(false);
  pauseButton.setEnabled(false);
}
/**
  * Callback for the pause button
  * Pause or resume the current simulation
  */
public void pauseSimulation() {
  if (sim.state == Simulator.RUNNING) {
    sim.state = Simulator.PAUSED;
    pauseButton.setLabel("Resume");
  } else {
    sim.state = Simulator.RUNNING;
    pauseButton.setLabel("Pause");
    runSimulation();
  }
}

public void paint(Graphics g) {
  super.paint(g);
  plot1.paint(g);
  plot2.paint(g);
  plot3.paint(g);
}

public void handle(Message message) {
  // Add a point
  // Use standart out for now
  if (message.command.equals("display")) {
    plot1.addPoint(0, sim.now() ,(double)SexualPerson.countSusceptible,!firstPoint);
    plot2.addPoint(0, sim.now() ,(double)
      SexualPerson.countInfectious,!firstPoint);
    plot3.addPoint(0, sim.now() ,(double)
      SexualPerson.countPartnerships,!firstPoint);
    firstPoint = false;
    repaint();
    try {
      Thread.currentThread().yield();
    } catch (Exception e) {
      e.printStackTrace();
    }
    if (true || DEBUG) System.out.println(""+ sim.now() + " " +
      SexualPerson.countSusceptible + " " +
      SexualPerson.countInfectious + " " +
      SexualPerson.countPartnerships);
    message.time += displayInterval;
    sim.insert(message);
```

```
    }

  }

  public boolean action(java.awt.Event e, Object arg) {
    if (e.target == startButton) {
      startSimulation();
    }
    if (e.target ==pauseButton) {
      pauseSimulation();
    }
    if (e.target ==stopButton) {
      stopSimulation();
    }
    return super.action(e,arg);
  }
}
```

Lesson 11: Coroutines and Threads

In SIMULA, which was an object-oriented simulation language long before either term was fashionable, the life cycle of objects was defined by blocks of code with ebedded wait statements. For example, a caricature of a human life might be described in SIMULA/DEMOS as

```
CLASS PERSON;
   BEGIN
      HOLD(20);
      GET_MARRIED();
      HOLD(.75);
      HAVE_KID();
      HOLD(2);
      HAVE_KID();
      HOLD(50);
      DIE();
   END;
END;
```

Smalltalk also provides its own mechanism for a block of code to suspend execution and resume at some later time. In Java and other languages with threads, this mechanism is even built on top of the operating system's notion of threads. C++Sim provides a C++ interface to Posix threads and other thread libraries. All of these solutions allow a modeler to develop models that are easier to read. Where you have thread support from the operating system you may even get automatic parallelization on multiple processor machines. Unfortunately, some of these solution require considerable computational overhead and are less portable than a pure event model. In particular, the thread support in Java only partially implemented, differs across platforms, and is in flux.

Epilogue: What Next?

These lessons have built only the absolutely essential parts of a simulation. If you were to develop this further, here are some of the areas that you might want to fill in.

- Faster event queue
- More random distributions
- User interface
- Ways to gather summaries of the many simulation entities
- Animation and other ways to monitor the simulation
- Coroutines
- Model builder
- Distributed computing

- Outcome estimation
- Comparison
- Sensitivity analysis
- Optimization, including parameter estimation

Bibliography

A Guide to Simulation, by P. Bratley, B. L. Fox and L. E. Schrage. Simulation Modeling and Analysis, by A. Law and D. Kelton
Simulation Model Design and Execution, by P. Fishwick
Discrete Systems Simulation, by B. Khoshnevis
Theory of Modeling and Simulation, by B. Ziegler
Nonuniform Random Variate Generation, by L. Devroye

This page was put together by Michael Altmann. Drop me a line at Michael@umn.edu with your comments.