Chapter 4
Simulation Programming with JavaSim

This chapter shows how simulations of some of the examples in Chap. 3 can be programmed in JavaSim. The goals of the chapter are to introduce JavaSim, and to hint at the experiment design and analysis issues that will be covered in later chapters.

4.1 JavaSim Overview

JavaSim is a collection of Java methods and classes that aid in developing discrete-event simulations. They are entirely open source and can be modified to suit the user. The JavaSim package consists of the JavaSim class, supporting classes for simulations, and examples of simulations written using JavaSim. The random-number and random-variate generation routines implemented in the class Rng are Java translations of the corresponding routines in simlib (Law 2007) which is written in `C`. JavaSim is designed to be easy to understand and use, but not necessarily efficient.

Here is a brief description of the methods in the JavaSim class:

`javaSimInit:` Initializes JavaSim for use, typically called before the start of each replication.
`schedule:` Schedules future events.
`schedulePlus:` Schedules future events and allows an object to be stored with the event.
`report:` Writes a result to a specific row and column of a JTable table.
`clearStats:` Clears certain statistics being recorded by JavaSim.

Here is a brief description of the methods in the Rng class:

`Rng:` Random number generator class. The constructor initializes the random-number generator; typically called only once in a simulation.
`expon:` Generates exponentially distributed random variates.
`uniform:` Generates uniformly distributed random variates.

`randomInteger:` Generates a random integer.
`erlang:` Generates Erlang distributed random variates.
`normal:` Generates normally distributed random variates.
`lognormal:` Generates lognormally distributed random variates.
`triangular:` Generates triangularly distributed random variates.

The random-variate generation methods take two types of arguments: parameters, and a random-number stream; the random-number stream is always the last argument. For instance

```
x = uniform(10, 45, 2)
```

generates random-variates that are uniformly distributed between 10 and 45, using stream 2.

As you already know, the pseudorandom numbers we use to generate random variates in simulations are essentially a long list. Random-number streams are just different starting places in this list, spaced far apart. The generator in JavaSim (which is a translation of the generator in Law 2007) has 100 streams. Calling the random-number generator constructor `Rng()` sets the initial position of each stream. Then each subsequent call to a variate-generation routine using stream # advances stream # to the next pseudorandom number. The need for streams is discussed in Chap. 7, but it is important to know that any stream is as good as any other stream in a well-tested generator.

Here is a brief description of the classes in JavaSim:

`Entity:` Object used to model transient items that flow through the system.
`FIFOQueue:` Object to hold entities in first-in-first-out order.
`Resource:` Object used to model a scarce quantity.
`EventNotice:` Object used to represent a future event.
`EventCalendar:` Data structure holding event notices in chronological order.
`CTStat:` Object used to record continuous-time statistics.
`DTStat:` Object used to record discrete-time statistics.

4.2 Simulating the $M(t)/M/\infty$ Queue

Here we consider the parking lot example of Sect. 3.1, a queueing system with time-varying car arrival rate, exponentially distributed parking time and an infinite number of parking spaces. The simulation program consists of the class initializations (Fig. 4.1), a main program and some event routines (Fig. 4.2), an initialization method (Fig. 4.3), a method to generate car arrivals (Fig. 4.4), and the support functionality provided by JavaSim. Two important aspects of JavaSim are illustrated by this model: event scheduling and collecting time-averaged statistics.

The key state variable that we need to track is the number of cars in the lot. There are two essential events that affect the state, the arrival of a car and the departure of a car, and a third event we will use to stop the simulation at a specific point in

time. The tricky part is that there can be an unlimited number of pending "departure" events; in fact, there are as many pending departure events as there are cars in the lot. Therefore, having a unique variable to represent the scheduled time of each pending event, as was used for the `TTF` example in Chap. 2, will not work.

To handle event scheduling, JavaSim has an event calendar class called `EventCalendar`, a method called `schedule` for putting events on the `EventCalendar` and a method called `remove` for extracting the next event from the calendar. An event in JavaSim is a Java object of class `EventNotice` which has (at least) two properties: `EventType` and `EventTime`. The statement

JavaSim.`schedule(name, increment)`

creates an `EventNotice`, assigns the character string `name` to the `EventType` property, assigns the value `clock + increment` to the `EventTime` property, and schedules the `EventNotice` on the `EventCalendar` in the correct chronological order. The method `JavaSim.calendarRemove()` extracts the chronologically next `EventNotice` from the `EventCalendar`, making its `EventType` and `EventTime` properties available to advance time and execute the proper event.

The simulation main program `MtMInf` in Fig. 4.2 illustrates how the event-related features of JavaSim work. The following statements, or ones very similar, will appear in all simulations using JavaSim:

```
EventNotice nextEvent;

nextEvent = javaSim.calendarRemove();
javaSim.setClock(nextEvent.getEventTime());
if (nextEvent.getEventType() == specifiedEventType1) {

}
else if (nextEvent.getEventType() == specifiedEventType2) {

}
...
else if(nextEvent.getEventType() == specifiedEventTypeN) {

}
```

Since our simulation will repeatedly remove the next `EventNotice` from `EventCalendar`, we need an object of this class to which to assign it; the statement `EventNotice nextEvent` provides this. The statement `nextEvent = javaSim.calendarRemove()` illustrates how the `remove` method extracts the next event, after which we advance the simulation `clock` to time `nextEvent.EventTime` and execute the event indicated by `nextEvent.EventType`.

```
/**
 * Example illustrating use of JavaSim for simulation of
 * M(t)/M/infinity Queue parking lot example. In this
 * version parking time averages 2 hours; the arrival
 * rate varies around 100 per hour; the lot starts empty,
 * and we look at a 24-hour period.
 * See JavaSim package for generic declarations and for
 * the supporting JavaSim class
 */

  // simulation variables and statistics
 private int     n;              // Number in queue
 private CTStat  queueLength;    // use to keep statistics on n
 private int     maxQueue;       // largest observed value of n
```

Fig. 4.1 Declarations for the parking lot simulation.

JavaSim's `schedule` method places events on the `EventCalendar`; for instance, the statement

```
JavaSim.Schedule("EndSimulation", 24)
```

creates an `EventNotice` of type EndSimulation to occur 24 hours from the time currently on `clock` (which is 0 at the beginning of the simulation). Notice that JavaSim requires the user to decide on the base time unit in the simulation and to use it consistently throughout. In this simulation the base time unit is hours.

The key state variable in this simulation is `n`, the current number of cars in the parking lot, and we want statistics on it. JavaSim contains a `CTStat` class that can be used to record time-averaged statistics. Here is how it works: First, we declare a new `CTStat` using the statement

```
Private CTStat queueLength;
```

as shown in the class declarations (Fig. 4.1). Second, the `CTStat` should be initialized using the statement

```
queueLength = new CTStat();
```

as shown in Fig. 4.3. Third, the `CTStat` can (and usually should) be added to a special collection called `TheCTStats`, as shown in Fig. 4.3. JavaSim reinitializes any `CTStat` in `TheCTStats` collection whenever `JavaSim.javaSimInit()` is executed, which is typically at the beginning of each replication. Next, whenever the value of the variable of interest changes, the `record` method of the `CTStat` is employed to record the change (which means it is called just after the change occurs); in this simulation the statement is `queueLength.record(n, javaSim.getClock())`, as shown in Fig. 4.2. Finally, the time-averaged value of the `CTStat` can be computed using the `Mean` method of the `CTStat`, as in `queueLength.mean(javaSim.getClock())`.

Notice that JavaSim also has a `report` method that writes a value or character string to a given row and column of a Java JTable. The syntax is

```
/**
 * Run the MTMInfinity simulation and output the results
 */

private void runSimulation() {

  EventNotice nextEvent;

  for(int reps = 0; reps < 1000; reps++) {
    n = 0;
    maxQueue = 0;
    javaSim.javaSimInit(); // initialize javaSim for each replication

    javaSim.schedule("Arrival", nspp(0));
    javaSim.schedule("EndSimulation", 24);

    do {
      nextEvent = javaSim.calendarRemove();
      javaSim.setClock(nextEvent.getEventTime());
      if (nextEvent.getEventType() == "Arrival") {
        arrival();
      } else if (nextEvent.getEventType() == "Departure") {
        departure();
      }
    } while(nextEvent.getEventType() != "EndSimulation");

    javaSim.report(queueLength.mean(javaSim.getClock()), reps + 1, 0);
    javaSim.report(maxQueue, reps + 1, 1);

  }

}
```

Fig. 4.2 Main program and event routines for the parking lot simulation.

```
JavaSim.report(value or string, row, column)
```

See Figs. 4.2 and 4.3.

Recall that the arrival rate (in cars per hour) to the parking lot was modeled by the function $\lambda(t) = 1000 + 100\sin(\pi t/12)$. To make the simulation execute more quickly for the purpose of this introduction, we change that rate to $\lambda(t) = 100 + 10\sin(\pi t/12)$, so that the arrival rate varies between 90 and 110 cars per hour, depending on the hour of the day t.

The method `nspp` shown in Fig. 4.4 generates interarrival times (time gaps between arrivals) from a nonstationary Poisson process with this arrival rate. The formal definition of a nonstationary Poisson process is a topic of Chap. 6. However, we provide an intuitive justification for the working of method `nspp` here:

A stationary Poisson process has times between arrivals that are exponentially distributed with a fixed rate λ (or equivalently a constant mean time between arrivals $1/\lambda$). The inverse cdf method for generating exponentially distributed random variates was described in Chap. 2.2.1. The maximum arrival rate for $\lambda(t)$ is 110 cars

```
/**
 * Initialize the simulation
 */

private void myInit() {
  // initialize the random number generator
  generator = new Rng();

  // initialize the simulation
  String simulationName = "M(t)/M/infinity";
  javaSim = new JavaSim(simulationName);

  // initialize the queue length CTStat
  queueLength = new CTStat();

  meanParkingTime = 2.0;
  javaSim.addCTStat(queueLength);

  // Write headings for the output reports
  javaSim.report("Average Number in Queue", 0, 0);
  javaSim.report("Maximum Number in Queue", 0, 1);

}
```

Fig. 4.3 Initializing the parking lot simulation.

per hour, so `nspp` generates possible arrivals using a stationary arrival process with rate $\lambda = 110$. To achieve the time-varying arrival rate, it only accepts a *possible* arrival at time t as an *actual* arrival with probability $\lambda(t)/\lambda$. Thus, a possible arrival that is to occur at a time when $\lambda(t) = 110$ will always be an actual arrival, while a possible arrival that is to occur when $\lambda(t) = 90$ will only have a $90/110$ probability of being an actual arrival. That this method, which is called "thinning," generates a nonstationary Poisson process with the desired rate is discussed in Chap. 6.

Fig. 4.5 shows a histogram of the 1000 daily averages of the number of cars in the parking lot obtained by running the simulation for 1000 replications; the overall average of these averages is 184.2 ± 0.3 cars, where the "± 0.3" part comes from a 95% confidence interval on the mean (confidence intervals are a subject of Chap. 7). Thus, the simulation provides a pretty precise estimate of the time-average mean number of cars that would be found in the (conceptually) infinite-size garage during a day.

The histogram shows that the average number of cars in the garage can vary substantially from day to day, so we certainly would not want to build a garage with a capacity of, say, 185 cars. Further, averaging over the day masks the largest number of cars in the garage during the day, and that number is more useful for selecting a finite capacity for the garage.

JavaSim provides no special support for the maximum statistic, but since we have access to everything in a JavaSim simulation we can easily record whatever we want. Here we define a variable `maxQueue` which is initialized to 0 at the beginning of

```
/**
 * This function implements thinning to generate
 * interarrival times from the
 * nonstationary Poisson arrival process
 * representing car arrivals. Time units
 * are minutes.
 *
 * @param stream
 *          Seed for the random number generator
 * @return interarrival time from nonstationary
 * Poisson arrival process
 */

private double nspp(int stream) {
  double possibleArrival =
  javaSim.getClock() + generator.expon(1.0 / 110.0, stream);

  while(generator.uniform(0, 1, stream)
  >= (100 + 10 *
  Math.sin(3.141593 * possibleArrival / 12.0)) / 110.0) {
    possibleArrival += generator.expon(1.0 / 110.0, stream);
  }

  return possibleArrival - javaSim.getClock();

}
```

Fig. 4.4 Method to generate interarrival times to the parking lot.

each replication, and is increased to the current value of `n` whenever `n` exceeds the previous value of `maxQueue`. See in particular the method `arrival` in Fig. 4.2.

Suppose that we want the parking lot to be of adequate size 99% of the time. Since we record the maximum size on 1000 replications, we could use the 990th value (sorted from smallest to largest) as the size of the garage, which turned out to be 263 cars in this simulation. Figure 4.6 shows the empirical cumulative distribution (ecdf) of the 1000 maximums recorded by the simulation. The ecdf treats each observed value as equally likely (and therefore as having probability $1/1000$), and plots the sorted maximum values on the horizontal axis and the cumulative probability of each observation on the vertical axis. The plot shows how the cumulative probability of 0.99 maps to the value of 263 cars, which might be a very reasonable capacity for the garage. Putting a confidence interval on this value is quite different than putting one on the mean, and will be discussed in Chap. 7. Without a confidence interval (or some measure of error) we cannot be sure if 1000 replications is really enough to estimate the 99th percentile of the maximum.

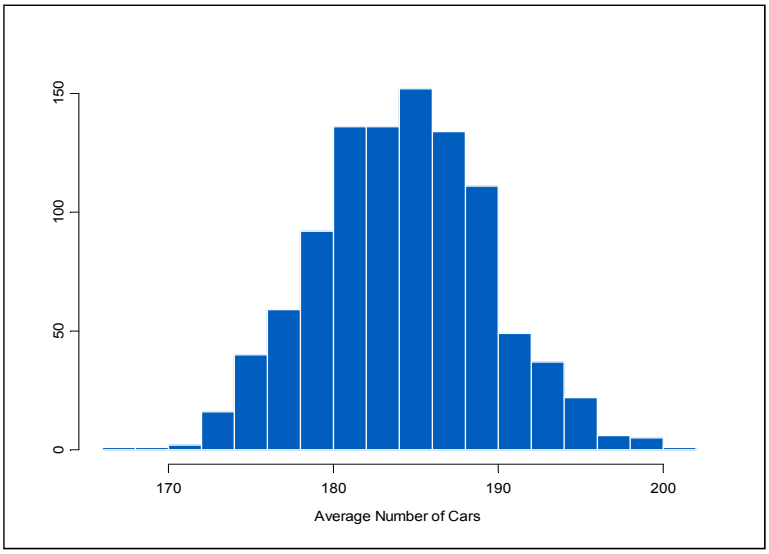

Fig. 4.5 Histogram of the daily average number of cars.

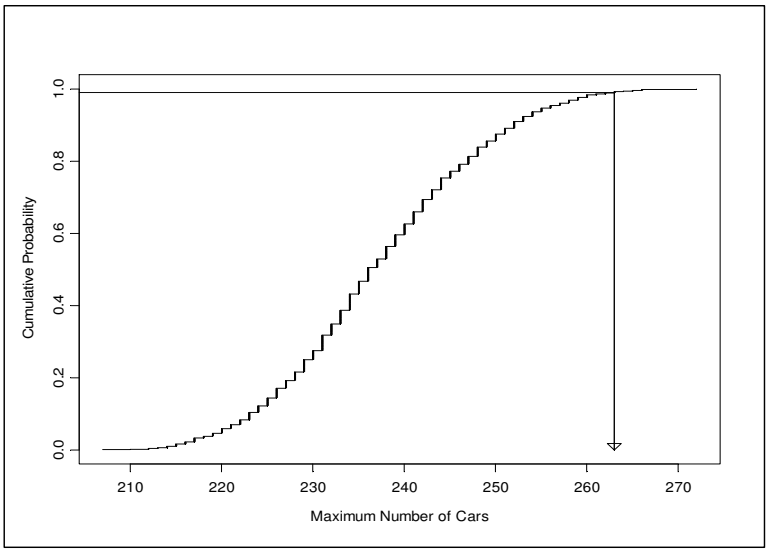

Fig. 4.6 Empirical cdf of the daily maximum number of cars in the parking lot.

4.2.1 Issues and Extensions

1. The $M(t)/M/\infty$ simulation presented here simulates 24 hours of parking lot operation, and treats each 24-hour period as as independent replication starting with

an empty garage. This only makes sense if the garage is emptied each day, for instance if the mall closes at night. Is the assumed arrival rate $\lambda(t)$ appropriate for a mall that closes at night?

2. Suppose that the parking lot serves a facility that is actually in operation 24 hours a day, seven days per week (that is, all the time). How should the simulation be initialized, and how long should the run length be in this case?
3. How could the simulation be initialized so that there are 100 cars in the parking lot at time 0?
4. When this example was introduced in Sect. 3.1, it was suggested that we size the garage based on the (Poisson) distribution of the number of cars in the garage at the point in time when the mean number in the garage was maximum. Is that what we did, empirically, here? If not, how is the quantity we estimated by simulation related to the suggestion in Sect. 3.1 (for instance, will the simulation tend to suggest a bigger or smaller garage than the analytical solution in Sect. 3.1)?
5. One reason that this simulation executes quite slowly when $\lambda(t) = 1000 + 100\sin(\pi t/12)$ is that the thinning method we used is very inefficient (lots of possible arrivals are rejected). Speculate about ways to make it faster.
6. For stochastic processes experts: Another reason that the simulation is slow when $\lambda(t) = 1000 + 100\sin(\pi t/12)$ is that there can be 1000 or more pending departure events on `EventCalendar` at any time, which means that scheduling a new event in chronological order involves a slow search. However, it is possible to exploit the memoryless property of the exponential distribution of parking time to create an equivalent simulation that has only two pending events (the next car arrival and next car departure) at any point in time. Describe how to do this.

4.3 Simulating the $M/G/1$ Queue

Here we consider the hospital example of Sect. 3.2, a queueing system with Poisson arrival process, some (as yet unspecified) service-time distribution, and a single server (either a receptionist or an electronic kiosk); in other words, an $M/G/1$ queue. Patient waiting time is the key system performance measure, and the long-run average waiting time in particular.

Recall that Lindley's Equation (3.3) provides a shortcut way to simulate successive customer waiting times:

$$\begin{aligned} Y_0 &= 0 \qquad X_0 = 0 \\ Y_i &= \max\{0, Y_{i-1} + X_{i-1} - A_i\},\ i = 1, 2, \ldots \end{aligned}$$

where Y_i is the ith customer's waiting time, X_i is that customer's service time, and A_i is the interarrival time between customers $i-1$ and i. Lindley's equation avoids the need for an event-based simulation, but is limited in what it produces (how would you track the time-average number of customers in the queue?). In this section we

will describe both recursion-based and event-based simulations of this queue, starting with the recursion.

4.3.1 Lindley Simulation of the $M/G/1$ Queue

To be specific, suppose that the mean time between arrivals is 1 minute, with the distribution being exponential, and the mean time to use the kiosk is 0.8 minutes (48 seconds), with the distribution being an Erlang-3. An Erlang-p is the sum of p i.i.d. exponentially distributed random variables, so an Erlang-3 with mean 0.8 is the sum of 3 exponentially distributed random variables each with mean $0.8/3$.

In Sect. 3.2 we noted that the waiting-time random variables $Y_1, Y_2, \ldots$ converge in distribution to a random-variable Y, and it is $\mu = \mathrm{E}(Y)$ that we will use to summarize the performance of the queueing system. We also noted that $\bar{Y}(m) = m^{-1}\sum_{i=1}^{m} Y_i$ converges with probability 1 to μ as the number of customers simulated m goes to infinity.

All of this suggests that we make a very long simulation run (large m) and estimate μ by the average of the observed waiting times $Y_1, Y_2, \ldots, Y_m$. But this is not what we will do, and here is why: Any m we pick is not ∞, so the waiting times early in the run—which will tend to be smaller than μ because the queue starts empty—will likely pull $\bar{Y}(m)$ down. To reduce this effect, we will let the simulation generate waiting times for a while (say d of them) before starting to actually include them in our average. We will still make m large, but our average will only include the last $m-d$ waiting times. That is, we will use as our estimator the truncated average

$$\bar{Y}(m,d) = \frac{1}{m-d}\sum_{i=d+1}^{m} Y_i. \tag{4.1}$$

In addition, we will not make a single run of m customers, but instead will make n replications. This yields n i.i.d. averages $\bar{Y}_1(m,d), \bar{Y}_2(m,d), \ldots, \bar{Y}_n(m,d)$ to which we can apply standard statistical analysis. This avoids the need to directly estimate the asymptotic variance γ^2, a topic we defer to later chapters.

Figure 4.7 shows a JavaSim simulation of the $M/G/1$ queue using Lindley's equation. In this simulation $m = 55{,}000$ customers, we discard the first $d = 5000$ of them, and make $n = 10$ replications. The ten replication averages are individually written to a JTable called "M/G/1 Lindley" and are displayed in Table 4.1.

Notice that the average waiting time is a bit over 2 minutes, and that Java, like all programming languages, displays a very large number of output digits. How many are really meaningful? A confidence interval is one way to provide an answer.

Since the across-replication averages are i.i.d., and since each across-replication average is itself the within-replication average of a large number of individual waiting times (50,000 to be exact), the assumption of independent, normally distributed output data is reasonable. This justifies a t-distribution confidence interval on μ. The key ingredient is $t_{1-\alpha/2,n-1}$, the $1-\alpha/2$ quantile of the t distribu-

```
package org.javasim.examples;

import org.javasim.JavaSim;
import org.javasim.Rng;

public class MG1Lindley {

  public static void main(String[] args) {
    int d;
    long m;
    double y, x, a;
    double sumY;

    Rng generator = new Rng();
    m = 55000;
    d = 5000;

    String simulationName = "M/G/1 Lindley";
    JavaSim javaSim = new JavaSim(simulationName);
    javaSim.report("Average Wait", 0, 0);

    for(int rep = 0; rep < 10; rep++) {
      y = 0.0;
      sumY = 0.0;

      for(int i = 0; i < d; i++) {
        a = generator.expon(1.0, 0);
        x = generator.erlang(3, 0.8, 1);
        y = Math.max(0, y + x - a);
      }

      for(int i = d; i < m; i++) {
        a = generator.expon(1.0, 0);
        x = generator.erlang(3, 0.8, 1);
        y = Math.max(0, y + x - a);
        sumY += y;
      }
      javaSim.report(sumY / ((double) m - d), rep + 1, 0);
    }

  }
}
```

Fig. 4.7 Simulation of the $M/G/1$ queue using Lindley's equation.

Table 4.1 Ten replications of the $M/G/1$ queue using Lindley's equation.

replication	$\bar{Y}(55000,5000)$
1	2.191902442
2	2.291913404
3	2.147858324
4	2.114346960
5	2.031447995
6	2.110924602
7	2.132711743
8	2.180662859
9	2.139610760
10	2.146212039
average	2.148759113
std dev	0.066748617

tion with $n-1$ degrees of freedom. If we want a 95% confidence interval, then $1-\alpha/2 = 0.975$, and our degrees of freedom are $10-1=9$. Since $t_{0.975,9} = 2.26$, we get $2.148759113 \pm (2.26)(0.066748617)/\sqrt{10}$ or $2.148759113 \pm 0.047703552$. This implies that we can claim with high confidence that μ is around 2.1 minutes, or we could give a little more complete information as 2.14 ± 0.05 minutes. Any additional digits are statistically meaningless.

Is an average of 2 minutes too long to wait? To actually answer that question would require some estimate of the corresponding wait to see the receptionist, either from observational data or a simulation of the current system. Statistical comparison of alternatives is a topic of Chap. 8.

4.3.2 Event-based Simulation of the $M/G/1$ Queue

The simulation program consists of some class-level declarations (Fig. 4.8), a main program (Fig. 4.9), some event routines (Fig. 4.10), an initialization method (Fig. 4.11), and the support functionality provided by JavaSim. Four JavaSim class objects and one method are illustrated by this model: `Entity`, `FIFOQueue`, `Resource` `DTStat` and `clearStats`. At a high level, here is what they do:

- `Entity` objects are used to model transient items, such as transactions or customers that pass through the system. They can have attributes (Java properties) that they carry with them; by default they have an attribute called `createTime` which is set to the value of `clock` when an `Entity` object is created. In this simulation the `Entity` objects represent patients or visitors.
- `FIFOQueue` is a Java collection, much like `EventCalendar`, that is used for holding `Entity` objects in first-in-first-out order; it also records time-average number in the queue statistics. In this simulation the queue represents the patients using or waiting for the kiosk.

```
/**
* Example illustrating use of JavaSim for simulation of
*  M/G/1 Queue.
* See JavaSim package for generic declarations and for
* the supporting JavaSim class
*/

 // parameters we may want to change
 private double  meanTBA;  // mean time between arrivals
 private double  meanST;   // mean service time
 private int     phases;   // number of phases in
                           // service distribution
 private double  runLength; // run length
 private double  warmUp;   // "warm-up" time

 // objects for simulation
 // these will usually be queues and statistics
 FIFOQueue       queue;    // customer queue
 DTStat          wait;     // discrete-time statistics
                           // on customer waiting
 Resource        server;   // server resource

 // simulation object and the random number generator
 private JavaSim javaSim;
 private Rng     generator;
```

Fig. 4.8 Declarations for the hospital simulation.

- `Resource` objects represent scarce quantities like workers, machines and computers that are needed (typically) to serve or process an `Entity` in some way; they also record the average number of busy resource units. In this simulation the `Resource` object represents the kiosk.
- `DTStat` is an object for recording discrete-time statistics; it is the companion to `CTStat` for continuous-time statistics. In this simulation a `DTStat` is used to record total waiting-time statistics, where we will define "total waiting time" to be the time from patient arrival until they are finished with the kiosk. Notice that this is (intentionally) different from the definition in Sect. 4.3.1, where the wait only included the time to reach the front of the line.
- `clearStats` is a method that reinitializes all statistical variables found in two collections, `TheCTStats` and `TheDTStats`. `FIFOQueue` and `Resource` objects each create a `CTStat` which is automatically added to `TheCTStats`. When the programmer creates a custom `CTStat` or `DTStat` then they must add it to the appropriate collection.

Figure 4.8 shows the declarations of the `FIFOQueue`, `DTStat` and `Resource` objects, specifically:

```
FIFOQueue queue;
DTStat wait;
```

```
/**
 * Run the M/G/1 simulation and output the results
 */

private void runSimulation() {

  EventNotice nextEvent;

  for(int reps = 0; reps < 10; reps++) {
    javaSim.javaSimInit();
    javaSim.schedule("Arrival", generator.expon(meanTBA, 0));
    javaSim.schedule("EndSimulation", runLength);
    javaSim.schedule("ClearIt", warmUp);

    do {
      nextEvent = javaSim.calendarRemove();
      javaSim.setClock(nextEvent.getEventTime());
      if (nextEvent.getEventType() == "Arrival") {
        arrival();
      } else if (nextEvent.getEventType() == "EndOfService") {
        endOfService();
      } else if (nextEvent.getEventType() == "ClearIt") {
        javaSim.clearStats();
      }

    } while(nextEvent.getEventType() != "EndSimulation");

    // write output report for each replication
    javaSim.report(wait.mean(), reps + 1, 0);
    javaSim.report(queue.mean(javaSim.getClock()), reps + 1, 1);
    javaSim.report(queue.numQueue(), reps + 1, 2);
    javaSim.report(server.mean(javaSim.getClock()), reps + 1, 3);

  }
}
```

Fig. 4.9 Main program for the hospital simulation.

```
Resource server;
```

These are declared as class members because there is only one of each and they will be referenced from many places in the simulation code.

This contrasts with the `Entity` objects that will be created and discarded as needed to represent patients coming and going. For instance, consider this statement in the method `arrival`:

```
queue.add(new Entity(javaSim.getClock()), javaSim.getClock());
```

The keyword `new` causes a new instance of the `Entity` class to be created. We can access its attributes using its public getter methods, as in `Entity.getCreateTime()`. The `add` method of the `FIFOQueue` object `queue` places `Entity` objects into the queue in order of arrival.

Moving into the `endOfService` event routine, the following two statements remove the first customer from the queue, use its `createTime` attribute to compute the total waiting time and record this value using the `DTStat` object `wait`. It is important to notice the *absence* of the keyword `new` in `Entity departingCustomer`; this means that `departingCustomer` is declared to be of type `Entity`, but a new `Entity` object is not created by the declaration statement.

```
Entity departingCustomer = (Entity) queue.remove(javaSim.getClock());
wait.record(javaSim.getClock() - departingCustomer.getCreateTime());
```

Before a `Resource` object can be used, its capacity (number of identical units) must be set. In the method `myInit` this is accomplished by using the object's `setUnits` method, `server.setUnits (1)`. If, for instance, there were 3 identical kiosks then this statement would be `server.setUnits (3)`. To make one (or more) units of a `Resource` busy, the `seize` method is employed, while idling a `Resource` is accomplished via the `free` method, as shown in the event methods.

In this simulation we are interested in long-run performance, so the length of a replication is determined by whatever we decide is long enough (which is not an easy decision, actually). When we used Lindley's equation it was natural to specify the replication length in terms of the number of customers simulated. However, in more complex simulations with many different outputs, it is far more common to specify the replication length by a stopping time T chosen so that it will be long enough for all of the outputs. Similarly, if we plan to discard data, then it is easier to specify a time T_d at which all statistics will be cleared.

To control the replication length by time, we schedule two events: an "EndSimulation" event scheduled to occur at time $T = 55{,}000$ minutes, and a statistics clearing event "ClearIt" that calls the method `clearStats` at time $T_d = 5000$ minutes. *Because the arrival rate of patients and visitors is* 1 *per minute, these times will approximately correspond to the run lengths of* 55,000 *and* 5000 *patients; however the actual number of patients will be random and vary from replication to replication.*

Let $\bar{Y}(T, T_d)$ be a replication average of all waiting times recorded between times T_d and T. Clearly $\bar{Y}(m, d)$ based on count, and $\bar{Y}(T, T_d)$ based on time, will have different statistical properties, but it is intuitively clear that both will be good estimators of μ if their arguments are fixed (not a function of the data) and large enough. Notice also that a run time and deletion time are ideal for continuous-time statistics like the time-average number in queue.

For this event-based simulation it is easy to record and report a number of statistics. `FIFOQueue`, `Resource` and `DTStat` objects all have `getMean` methods for reporting average values, which for this simulation are `queue.getMean`, `server.getMean` and `wait.getMean`, respectively. In addition, the `FIFOQueue` objects have a `numQueue` method to deliver the current number of entities in the queue; here we use it to report the number of patients in the queue at time 55,000 when the replication ends. Table 4.2 shows the results from 10 replications, along with the overall averages and 95% confidence interval halfwidths.

```
/**
 * Arrival event
*/

private void arrival() {
  // schedule next arrival
  javaSim.schedule("Arrival", generator.expon(meanTBA, 0));

  // process the newly arriving customer

  queue.add(new Entity(javaSim.getClock()), javaSim.getClock());

  // If server is not busy, start service by seizing the server
  if (server.getBusy() == 0) {
    server.seize(1, javaSim.getClock());
    javaSim.schedule
    ("EndOfService", generator.erlang(phases, meanST, 1));
  }

}

/**
 * End of service event
 */

private void endOfService() {
  // remove departing customer from queue and record wait time

  Entity departingCustomer =
  (Entity) queue.remove(javaSim.getClock());
  wait.record
  (javaSim.getClock() - departingCustomer.getCreateTime());

  // Check to see if there is another customer;
  // if yes start service otherwise free the server

  if (queue.numQueue() > 0) {
    javaSim.schedule
    ("EndOfService", generator.erlang(phases, meanST, 1));
  } else {
    server.free(1, javaSim.getClock());
  }
}
```

Fig. 4.10 Event routines for the hospital simulation.

```
/**
 * Initialize the simulation
 */

private void myInit() {

  // initialize the simulation
  String simulationName = "M/G/1 Simulation";
  javaSim = new JavaSim(simulationName);

  // initialize the simulation objects
  wait = new DTStat();
  queue = new FIFOQueue(javaSim);
  server = new Resource(javaSim);
  server.setUnits(1); // set the number of servers to 1


  // initialize the random number generator
  generator = new Rng();


  meanTBA = 1.0;
  meanST = 0.8;
  phases = 3;
  runLength = 55000.0;
  warmUp = 5000.0;

  // Add queues, resources and statistics that need to be
  // initialized between replications to the class collections

  javaSim.addDTStat(wait);
  javaSim.addQueue(queue);
  javaSim.addResource(server);

  // write headings for the output reports
  javaSim.report("Average Wait", 0, 0);
  javaSim.report("Average Number in Queue", 0, 1);
  javaSim.report("Number Remaining in Queue", 0, 2);
  javaSim.report("Server Utilization", 0, 3);

}
```

Fig. 4.11 Initializing the hospital simulation.

Again there are meaningless digits, but the confidence intervals can be used to prune them. For instance, for the mean total wait we could report 2.93 ± 0.05 minutes. How does this statistic relate to the 2.14 ± 0.05 minutes reported for the simulation via Lindley's equation? Mean total time in the kiosk system (which is what the event-based simulation estimates) consists of mean time waiting to be served (which is what the Lindley simulation estimates) plus the mean service time (which

Table 4.2 Ten replications of the $M/G/1$ queue using the event-based simulation.

Rep	Total Wait	Queue	Remaining	Utilization
1	2.996682542	3.01742345	1	0.806654823
2	3.103155842	3.127773149	7	0.807276539
3	2.951068607	2.948352414	2	0.799341091
4	2.848547497	2.846673097	0	0.794701908
5	2.908913572	2.900437432	2	0.798615617
6	2.895622648	2.896900635	3	0.801983001
7	2.909777649	2.901219722	0	0.798658678
8	2.914666297	2.908612119	4	0.795440658
9	2.922193762	2.923535588	0	0.799157017
10	2.873148311	2.862172885	0	0.799143690
average	2.932377673	2.933310049	1.9	0.800097302
stdev	0.07227642	0.082882288	2.282785822	0.004156967
95% ci	0.051654132	0.059233879	1.631449390	0.002970879

we know to be 0.8 minutes). So it is not surprising that these two estimates differ by about 0.8 minutes.

4.3.3 Issues and Extensions

1. In what situations does it make more sense to compare the simulated kiosk system to simulated data from the current receptionist system rather than real data from the current receptionist system?
2. It is clear that if all we are interested in is mean waiting time, defined either as time until service begins or total time including service, the Lindley approach is superior (since it is clearly faster, and we can always add in the mean service time to the Lindley estimate). However, if we are interested in the distribution of total waiting time, then adding in the mean service time does not work. How can the Lindley recursion be modified to simulate total waiting times?
3. How can the event-based simulation be modified so that it also records waiting time until service begins?
4. How can the event-based simulation be modified to clear statistics after exactly 5000 patients, and to stop at exactly 55,000 patients?
5. The experiment design method illustrated in the event-based simulation is often called the "replication-deletion" method. If we only had time to generate 500,000 waiting times, what issues should be considered in deciding the values of n (replications), m (run length) and d (deletion amount)? Notice that we must have $nm = 500{,}000$, and only $n(m-d)$ observations will be used for estimating μ.
6. An argument against summarizing system performance by long-run measures is that no system stays unchanged forever (55,000 patients is approximately 38 24-hour days during which time there could be staff changes, construction or

emergencies), so a measure like μ is not a reflection of reality. The counter argument is that it is difficult, if not impossible, to model all of the detailed changes that occur over any time horizon (even the time-dependent arrival process in the $M(t)/M/\infty$ simulation is difficult to estimate in practice), so long-run performance at least provides an understandable summary measure ("If our process stayed the same, then over the long run...."). Also, it is often mathematically easier to obtain long-run measures than it is to estimate them by simulation (since simulations have to stop). Considering these issues, what sort of analysis makes the most sense for the hospital problem?

4.4 Simulating the Stochastic Activity Network

Here we consider the construction example of Sect. 3.4 which is represented as a stochastic activity network (SAN). Recall that the time to complete the project, Y, can be represented as

$$Y = \max\{X_1 + X_4, X_1 + X_3 + X_5, X_2 + X_5\}$$

where X_i is the duration of the ith activity. This simple representation requires that we enumerate all paths through the SAN, so that the project completion time is the longest of these paths. Path enumeration itself can be time consuming, and this approach does not easily generalize to projects that have resources shared between activities, for instance. Therefore, we also present a discrete-event representation which is more complicated, but also more general.

4.4.1 Maximum Path Simulation of the SAN

Figure 4.12 shows a JavaSim implementation of the algorithm displayed in Sect. 3.4 and repeated here:

1. set $s = 0$
2. repeat n times:
   a. generate $X_1, X_2, \ldots, X_5$
   b. set $Y = \max\{X_1 + X_4, X_1 + X_3 + X_5, X_2 + X_5\}$
   c. if $Y > t_p$ then set $s = s + 1$
3. estimate θ by $\widehat{\theta} = s/n$

Since $\Pr\{Y \leq t_p\}$ is known for this example (see Eq. (3.12)), the true $\theta = \Pr\{Y > t_p\} = 0.165329707$ when $t_p = 5$ is also computed by the program so that we can compare it to the simulation estimate. Of course, in a practical problem we would

not know the answer, and we would be wasting our time simulating it if we did. Notice that all of the digits in this probability are correct—assuming that the numerical functions in Java did their job—although certainly not practically useful.

The simulation estimate turns out to be $\widehat{\theta} = 0.163$. A nice feature of a probability estimate that is based on i.i.d. outputs is that an estimate of its standard error is easily computed:

$$\widehat{\text{se}} = \sqrt{\frac{\widehat{\theta}(1-\widehat{\theta})}{n}}.$$

Thus, $\widehat{\text{se}}$ is approximately 0.011, and the simulation has done its job since the true value θ is well within $\pm 1.96\,\widehat{\text{se}}$ of $\widehat{\theta}$. This is a reminder that simulations do not deliver *the answer*, like Eq. (3.12), but do provide the capability to estimate the simulation error, and to reduce that error to an acceptable level by increasing the simulation effort (number of replications).

4.4.2 Discrete-event Simulation of the SAN

This section uses more advanced Java constructs than any other part of the book and may be skipped without loss of continuity.

As noted in Sect. 3.4, we can think of the completion of a project activity as an event, and when all of the inbound activities $\mathscr{I}(j)$ to a milestone j are completed then the outbound activities $i \in \mathscr{O}(j)$ can be scheduled, where the destination milestone of activity i is $\mathscr{D}(i)$. Thus, the following generic milestone event is the only one needed:

event milestone (activity ℓ inbound to node j)
$\mathscr{I}(j) = \mathscr{I}(j) - \ell$
if $\mathscr{I}(j) = \emptyset$ then
    for each activity $i \in \mathscr{O}(j)$
        schedule milestone(activity i inbound to node $\mathscr{D}(i)$ to occur X_i time units later)
end if

Of course, this approach shifts the effort from enumerating all of the paths through the SAN to creating the sets $\mathscr{I}, \mathscr{O}, \mathscr{D}$, but these sets have to be either explicitly or implicitly defined to define the project itself. The key lesson from this example, which applies to many simulations, is that it is possible to program a single event routine to handle many simulation events that are conceptually distinct, and this is done by passing event-specific information to the event routine. In this case we need to pass the inbound activity and the target node, and since this information is needed when the event is executed, not when it is scheduled, we need to store it with the event notice. To do this, we will create a new class module and make use of a feature of the `EventNotice` class module.

```
public class SANMax {

  public static void main(String[] args) {
    double n, c, tp, y, theta;
    double[] x = new double[5];

    Rng generator = new Rng();
    String simulationName = "Stochastic Activity Network";
    JavaSim javaSim = new JavaSim(simulationName);
    javaSim.report("Pr{Y > tp}", 0, 0);
    javaSim.report("True Theta", 0, 1);

    n = 1000.0;
    c = 0.0;
    tp = 5.0;

    for(int rep = 0; rep < n; rep++) {
      for(int i = 0; i < 5; i++) {
        x[i] = generator.expon(1.0, 6);
      }
      y = Math.max(Math.max(x[0] + x[3], x[0] + x[2] + x[4]),
      x[1] + x[4]);

      if (y > tp) {
        c++;
      }
    }

    javaSim.report(c/n, 1, 0);

    theta = 1.0 -
    ((tp * tp / 2.0 - 3.0 * tp - 3.0) *
     Math.exp(-2.0 * tp) +
    (-tp * tp / 2.0 - 3.0 * tp + 3.0) *
    Math.exp(-tp) + 1.0 - Math.exp(-3.0 * tp));

    javaSim.report(theta, 1, 1);

  }
}
```

Fig. 4.12 Simulation of the SAN as the maximum path through the network.

We define a new class module `Activity` which has two properties: `WhichActivity` and `WhichNode`:

```
/**
 * Object to model an activity-destination node pair
 */
private int whichActivity;
private int whichNode;
```

This object represents an activity, which has a number $0, 1, \ldots, 4$, and also a destination node, which we will number with $j = 0$ for a, $j = 1$ for b, $j = 2$ for c and $j = 3$ for d.

When we schedule an activity to be completed at some time in the future, we will associate an `Activity` with the `EventNotice` using its `whichObject` property; the `EventNotice` class module is reproduced below:

```
public class EventNotice {
  private double eventTime;
  private String eventType;
  private Object whichObject;

  /**
   * Event with time and type.
   *
   * @param eventTime
   *          Time of event
   * @param eventType
   *          String representing event
   */

  public EventNotice(double eventTime, String eventType) {
    this.eventTime = eventTime;
    this.eventType = eventType;
  }

  /**
   * Event with time, type and an object.
   *
   * @param eventTime
   *          Time of event
   * @param eventType
   *          String representing event
   * @param whichObject
   *          object associated with event
   */

  public EventNotice(double eventTime,
  String eventType, Object whichObject) {
    this.eventTime = eventTime;
    this.eventType = eventType;
    this.whichObject = whichObject;
  }

  public double getEventTime() {
    return eventTime;
```

```
}

public String getEventType() {
  return eventType;
}

public Object getWhichObject() {
  return whichObject;
}

// Add additional problem specific attributes here
```

Perhaps the most complex feature of this simulation is how nodes are represented as lists of lists of Java ArrayLists. An `ArrayList<Type>` is a Java data structure, very much like a one-dimensional array, except that an `ArrayList<Type>` contains objects of class Type and Java provides methods for adding, removing and referencing elements of an ArrayList. Unlike arrays, `ArrayList` dynamically change size as elements are added or removed. In this case, each element of the list `nodes.get(i).get(j)` is a list of activities. If $i = 0$ it is inbound activities, and if $i = 1$ it is outbound activities, for node j. Thus, `nodes.get(0).get(j)` plays the role of the set $\mathscr{I}(j)$, while `nodes.get(1).get(j)` corresponds to the set $\mathscr{O}(j)$. Another ArrayList called `Destination` associates each activity with its destination and plays the role of the set $\mathscr{D}$. The key initializations that define the SAN take place in the method `sanInit`, shown in Fig. 4.14.

One other Java feature that we use extensively in this example is the ability to define constants, meaning variables whose values are fixed once and for all in the simulation. This is done to make the simulation more readable. For instance,

```
private final int a     = 0;
private final int b     = 1;
private final int c     = 2;
private final int d     = 3;
private final int inTo  = 0;
private final int outOf = 1;
```

By doing this we avoid the need to remember that node c corresponds to the number 2, and what index corresponds to incoming vs. outgoing activities to a node.

Notice (see Fig. 4.13) that the simulation ends when there are no additional activities remaining to be completed. This can be checked via `JavaSim.calendarN()`, a method that provides the number of events currently on the event calendar.

A difference between this implementation of the SAN simulation and the one in Sect. 4.4.1 is that here we write out the actual time the project completes on each replication. By doing so, we can estimate $\Pr\{Y > t_p\}$ for any value of t_p by sorting the data and counting how many out of 1000 replications were greater than t_p. Figure 4.16 shows the empirical cdf of the 1000 project completion times, which is the simulation estimate of Eq. (3.12).

```
/* SAN Simulation using a discrete-event approach */
// Nodes is a multi-dimensional ArrayList of ArrayList,
// where each ArrayListis a list of inbound or outbound
// activities to that node For Nodes(i, j) = inbound i=1
// or outbound i=2 node j = 1 for a, j = 2 for b,
// j = 3 for c, j = 4 for d.
private ArrayList<ArrayList<ArrayList<Integer>>> nodes;
private List<Integer>                            destination;
// simulation object and the random number generator
private JavaSim                                  javaSim;
private Rng                                      generator;
// constants
private final int                                a     = 0;
private final int                                b     = 1;
private final int                                c     = 2;
private final int                                d     = 3;
private final int                                inTo  = 0;
private final int                                outOf = 1;
/* Run the SAN simulation and report output */
private void runSimulation() {
  EventNotice nextEvent;
  Activity thisActivity;
  for(int rep = 0; rep < 1000; rep++) {
    javaSim.javaSimInit();
    sanInit(); // initializes the activities in the SAN
    milestone(0, a); // causes outbound activities of
    // node a to be scheduled
    do {
      nextEvent = javaSim.calendarRemove();
      javaSim.setClock(nextEvent.getEventTime());
      thisActivity = (Activity) nextEvent.getWhichObject();
      milestone(thisActivity.getWhichActivity(),
      thisActivity.getWhichNode());
    } while(javaSim.calendarN() > 0); // stop when event
    // calendar is empty
    javaSim.report(javaSim.getClock(), rep + 1, 0);
  }
}
/* Initialize the simulation */
private void myInit() {
  // initialize the random number generator
  generator = new Rng();
  // initialize the simulation
  String simulationName = "Discrete-Event Stochastic Activity Network";
  javaSim = new JavaSim(simulationName);
  javaSim.report("Completion Time", 0, 0);
}
```

Fig. 4.13 Main program for the discrete-event SAN simulation.

```
/* Initialize the activities in the SAN */
private void sanInit() {
  // destinations
  // destination[i] corresponds to the destination of activity i
  // initialize the destination
  destination = new ArrayList<Integer>(5);
  // initialize the nodes
  nodes = new ArrayList<ArrayList<ArrayList<Integer>>>();
  for(int i = 0; i < 2; i++) {
    nodes.add(new ArrayList<ArrayList<Integer>>());
  }
  for(ArrayList<ArrayList<Integer>> node : nodes) {
    for(int j = 0; j < 4; j++) {
      node.add(new ArrayList<Integer>());
    }
  }
  destination.add(b);
  destination.add(c);
  destination.add(c);
  destination.add(d);
  destination.add(d);
  List<Integer> inbound = new ArrayList<Integer>();
  List<Integer> outbound = new ArrayList<Integer>();
  // node a
  outbound.add(0);
  outbound.add(1);
  nodes.get(inTo).get(a).addAll(inbound);
  nodes.get(outOf).get(a).addAll(outbound);
  inbound.clear();
  outbound.clear();
  // node b
  inbound.add(0);
  outbound.add(2);
  outbound.add(3);
  nodes.get(inTo).get(b).addAll(inbound);
  nodes.get(outOf).get(b).addAll(outbound);
  inbound.clear();
  outbound.clear();
  // node c
  inbound.add(1);
  inbound.add(2);
  outbound.add(4);
  nodes.get(inTo).get(c).addAll(inbound);
  nodes.get(outOf).get(c).addAll(outbound);
  inbound.clear();
  outbound.clear();
  // node d
  inbound.add(3);
  inbound.add(4);
  nodes.get(inTo).get(d).addAll(inbound);
  nodes.get(outOf).get(d).addAll(outbound);
  inbound.clear();
  outbound.clear();
}
```

Fig. 4.14 Network description for the discrete-event SAN simulation.

```
/**
 * Schedule a milestone, an activity 'actIn' inbound to node
 * 'node' to occur x
 * ~ expon(1) time units later
 *
 * @param actIn
 *          Inbound activity
 * @param node
 *          Node at which mile stone happens
 */

private void milestone(int actIn, int node) {
  int m;
  ArrayList<Integer> inbound =
    new ArrayList<Integer>(nodes.get(inTo).get(node));
  ArrayList<Integer> outbound =
    new ArrayList<Integer>(nodes.get(outOf).get(node));
  m = inbound.size();

  for(int incoming = 0; incoming < m; incoming++) {
    if (inbound.get(incoming) == actIn) {
      inbound.remove(incoming);
      break;
    }
  }
  nodes.get(inTo).get(node).clear();
  nodes.get(inTo).get(node).addAll(inbound);

  if (inbound.isEmpty()) {
    m = outbound.size();
    for(int actOut = 0; actOut < m; actOut++) {
      Activity thisActivity = new Activity();
      thisActivity.setWhichActivity(outbound.get(0));
      thisActivity.setWhichNode(destination.get(outbound.get(0)));
      javaSim.schedulePlus
        ("Milestone", generator.expon(1, 0), thisActivity);
      outbound.remove(0);
    }
  }

}
```

Fig. 4.15 Milestone event for the discrete-event SAN simulation.

4.4.3 Issues and Extensions

1. In real projects there are not only activities, but also limited and often shared resources that are needed to complete the activities. Further, there may be specific resource allocation rules when multiple activities contend for the same resource. How might this be modeled in JavaSim?

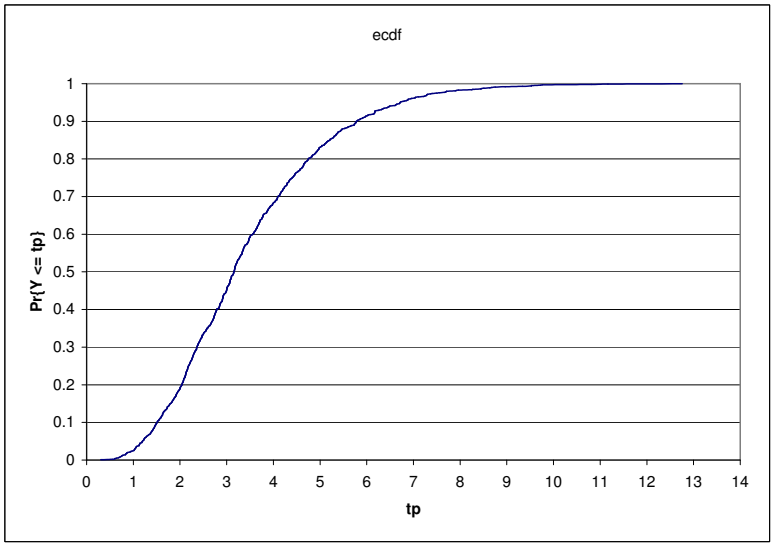

Fig. 4.16 Empirical cdf of the project completion times.

2. Time to complete the project is an important overall measure, but at the planning stage it may be more important to discover which activities or resources are the most critical to on-time completion of the project. What additional output measures might be useful for deciding which activities are "critical?"

4.5 Simulating the Asian Option

Here we consider estimating the value of an Asian option

$$v = \mathrm{E}\left[\mathrm{e}^{-rT}(\bar{X}(T) - K)^{+}\right]$$

as described in Sect. 3.5, where the maturity is $T = 1$ year, the risk-free interest rate is $r = 0.05$ and the strike price is $K = \$55$. The underlying asset has an initial value of $X(0) = \$50$ and the volatility is $\sigma^2 = (0.3)^2$. Recall that the key quantity is

$$\bar{X}(T) = \frac{1}{T}\int_0^T X(t)\,\mathrm{d}t$$

the time average of a continuous-time, continuous-state geometric Brownian motion process which we cannot truly simulate on a digital computer. Thus, we approximate it by dividing the interval $[0, T]$ into m steps of size $\Delta t = T/m$ and using the discrete approximation

$$\widehat{\bar{X}(T)} = \frac{1}{m}\sum_{i=1}^{m} X(i\Delta t).$$

This makes simulation possible, since

$$X(t_{i+1}) = X(t_i)\exp\left\{\left(r - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i}Z_{i+1}\right\}$$

for any increasing sequence of times $\{t_0, t_1, \ldots, t_m\}$, where $Z_1, Z_2, \ldots, Z_m$ are i.i.d. $N(0,1)$.

Figure 4.17 is JavaSim code that uses $m = 32$ steps in the approximation, and makes 10,000 replications to estimate v. Discrete-event structure would slow execution without any obvious benefit, so a simple loop is used to advance time. The value of the option from each replication is written to a JTable for post-simulation analysis.

The estimated value of v is \$2.20 with a relative error of just over 2% (recall that the relative error is the standard error divided by the mean). As the histogram in Fig. 4.18 shows, the option is frequently worthless (approximately 68% of the time), but the average payoff, conditional on the payoff being positive, is approximately \$6.95.

4.6 Case Study: Service Center Simulation

This section presents a simulation case based on a project provided by a former student. While still relatively simple, it is more complex than the previous stylized examples, and the answer is not known without simulating. The purpose of this section is to illustrate how one might attack simulation modeling and programming for a realistic problem.

Example 4.1 (Fax Center Staffing).

A service center receives faxed orders throughout the day, with the rate of arrival varying hour by hour. The arrivals are modeled by a nonstationary Poisson process with the rates shown in Table 4.3.

A team of Entry Agents select faxes on a first-come-first-served basis from the fax queue. Their time to process a fax is modeled as normally distributed with mean 2.5 minutes and standard deviation 1 minute. There are two possible outcomes after the Entry Agent finishes processing a fax: either it was a simple fax and the work on it is complete, or it was not simple and it needs to go to a Specialist for further processing. Over the course of a day, approximately 20% of the faxes require a Specialist. The time for a Specialist to process a fax is modeled as normally distributed with mean 4.0 minutes and standard deviation 1 minute.

Minimizing the number of staff minimizes cost, but certain service-level requirements much be achieved. In particular, 96% of all simple faxes should be completed within 10 minutes of their arrival, while 80% of faxes requiring a Specialist should

```
public class AsianOption {

  public static void main(String[] args) {
    int replications = 10000;
    double maturity = 1.0;
    int steps = 32;
    double sigma = 0.3;
    double interestRate = 0.05;
    double initialValue = 50.0;
    double strikePrice = 55.0;
    double interval = maturity / (double) steps;
    double sigma2 = sigma * sigma / 2.0;

    double x;
    double sum;
    double z;
    double value;

    // random number generator for the simulation
    Rng generator = new Rng();

    // Simulation object for the simulations
    String simulationName = "Asian Option";
    JavaSim simObject = new JavaSim(simulationName);

    simObject.report("Option Value", 0, 0);

    for(int i = 0; i < replications; i++) {
      sum = 0.0;
      x = initialValue;
      for(int j = 0; j < steps; j++) {
        z = generator.normal(0, 1, 11);
        x = x * Math.exp((interestRate - sigma2)
        * interval + sigma * Math.sqrt(interval) * z);
        sum = sum + x;
      }
      value = Math.exp(-interestRate * maturity)
      * Math.max(sum / (double) steps - strikePrice, 0.0);
      simObject.report(value, i + 1, 0);
    }

  }
}
```

Fig. 4.17 Java Simulation of the Asian option problem.

also be completed (by both the Entry Agent and the Specialist) within 10 minutes of their arrival.

The service center is open from 8 AM to 4 PM daily, and it is possible to change the staffing level at 12 PM. Thus, a staffing policy consists of four numbers: the number of Entry Agents and Specialists before noon, and the number of Entry Agents

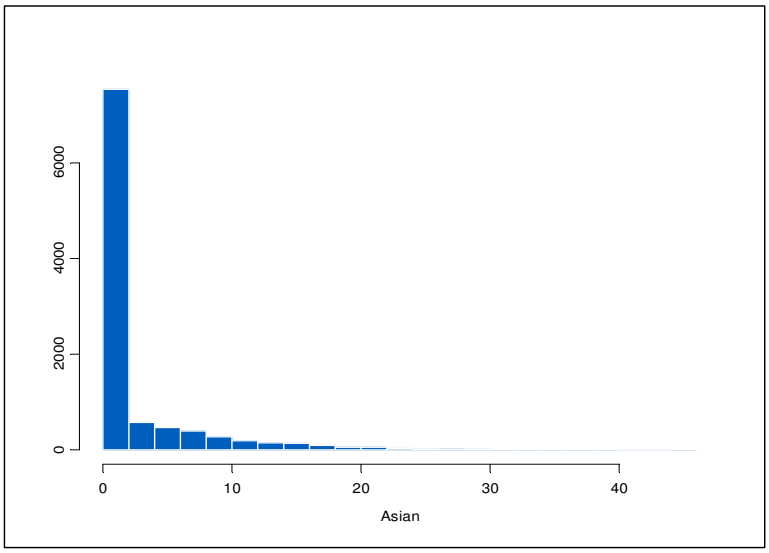

Fig. 4.18 Histogram of the realized value of the Asian option from 10,000 replications.

Table 4.3 Arrival rate of faxes by hour.

Time	Rate (faxes/minute)
8 AM–9 AM	4.37
9 AM–10 AM	6.24
10 AM–11 AM	5.29
11 AM–12 PM	2.97
12 PM–1 PM	2.03
1 PM–2 PM	2.79
2 PM–3 PM	2.36
3 PM–4 PM	1.04

and Specialists after noon. Any fax that starts its processing before noon completes processing by that same agent before the agent goes off duty; and faxes in the queues at the end of the day are processed before the agents leave work and therefore are not carried over to the next day.

The first step in building any simulation model is deciding what question or questions that the model should answer. Knowing the questions helps identify the system performance measures that the simulation needs to estimate, which in turn drives the scope and level of detail in the simulation model.

The grand question for the service center is, what is the minimum number of Entry Agents and Specialists needed for both time periods to meet the service-level requirements? Therefore, the simulation must at least provide an estimate of the percentage of faxes of each type entered within 10 minutes, given a specific staff assignment.

Even when there seems to be a clear overall objective (minimize the staff required to achieve the service-level requirement), we often want to consider trade offs around that objective. For instance, if meeting the requirement requires a staff that is so large that they are frequently underutilized, or if employing the minimal staff means that the Entry Agents or Specialists frequently have to work well past the end of the day, then we might be willing to alter the service requirement a bit. Statistics on the number and the time spent by faxes in queue, and when the last fax of each day is actually completed, provide this information. Including additional measures of system performance, beyond the most critical ones, makes the simulation more useful.

Many discrete-event, stochastic simulations involve entities that dynamically flow through some sort of queueing network where they compete for resources. In such simulations, identifying the entities and resources is a good place to start the model. For this service center the faxes are clearly the dynamic entities, while the Entry Agents and Specialists are resources. The fax machines themselves might also be considered a resource, especially if they are heavily utilized or if outgoing as well as incoming faxes use the same machines. It turns out that for this service center there is a bank of fax machines dedicated to incoming faxes, so it is reasonable to treat the arrival of faxes as an unconstrained external arrival process. This fact was not stated in the original description of the problem; follow-up questions are often needed to fully understand the system of interest.

Whenever there are scarce resources, queues may form. Queues are often first-in-first-out, with one queue for each resource, as they are in this service center. However, queues may have priorities, and multiple queues may be served by the same resource, or a single queue may feed multiple resources. Queueing behavior is often a critical part of the model.

When the simulation involves entities flowing through a network of queues, then there can be two types of arrivals: arrivals from outside of the network and arrivals internal to the network. Outside arrivals are like those we have already seen in the $M(t)/M/\infty$ and $M/G/1$ examples. Internal arrivals are departures from one queue that become arrivals to others. How these are modeled depends largely on whether the departure from one queue is an immediate arrival to the next—in which case the departure and arrival events are effectively the same thing—or whether there is some sort of transportation delay—in which case the arrival to the next queue should be scheduled as a distinct event. For the service center the arrival of faxes to the Entry Agents is an outside arrival process, while the 20% of faxes that require a Specialist are internal arrivals from the Entry Agents to the Specialists.

Critical to experiment design is defining what constitutes a replication. Replications should be independent and identically distributed. Since the service center does not carry faxes over from one day to the next, a "day" defines a replication. If faxes did carry over, but all faxes are cleared weekly, then a replication might be defined by a work week. However, if there is always significant carry over from one day to the next, then a replication might have to be defined arbitrarily.

The work day at the service center is eight hours; however the staff does not leave until all faxes that arrive before 4 PM are processed. If we defined a replication to be

exactly eight hours then we could be fooled by a staffing policy that allows a large queue of faxes to build up toward the end of the day, since the entry of those faxes would not be included in our statistics. To model a replication that ends when there is no additional work remaining, we will cut off the fax arrivals at 4 PM and then end the simulation when the event calendar is empty. This works because idle Entry Agents and Specialists will always take a fax from their queue if one is available.

Rather than walk through the JavaSim code line by line, we will point out some highlights to facilitate the reader's understanding of the code.

Figure 4.19 shows the class declarations for the service center simulation. Of particular note are the two `DTStat` statements defining `regular10` and `special10`. These will be used to obtain the fraction of regular and special faxes that are processed within the 10-minute requirement by recording a 1 for any fax that meets the requirement, and a 0 otherwise. The mean of these values is the desired fraction.

```
/**
 * FaxCenter Simulation
 */

// parameters we may want to change
private double meanRegular; // mean entry time regular faxes
private double varRegular; // variance entry time regular faxes
private double meanSpecial; // mean entry time special faxes
private double varSpecial; // variance entry time special faxes
private double runLength; // length of the working day
private int numAgents; // number of regular agents
private int numSpecialists; // number of special agents
private int numAgentsPM; // number of regular agents after noon
private int numSpecialistsPM; // number of special agents after noon


// class objects needed for simulation
private FIFOQueue regularQ; // queue for all faxes
private FIFOQueue specialQ; // queue for special faxes
private DTStat regularWait; // discrete-time statistics on fax waiting
private DTStat specialWait; // discrete-time statistics on special fax
// waiting
private DTStat regular10; // discrete-time statistics on < 10
// minutes threshold
private DTStat special10; // discrete-time statistics on < 10
// minutes threshold
private Resource agents; // entry agents resource
private Resource specialists; // specialists resource
private double[] aRate; // arrival rates
private double maxRate; // maximum arrival rate
private double period; // period for which arrival rate stays
// constant
private int nPeriods; // number of periods in a "day"

// simulation object and the random number generator
private JavaSim javaSim;
private Rng generator;
```

Fig. 4.19 Class declarations for service center simulation.

The main program for the simulation is in Fig. 4.20. Of particular importance is the condition that ends the main simulation loop:

```
while(javaSim.calendarN() > 0)
```

The `calendarN()` method of `JavaSim` returns the current number of pending events. When the event calendar is empty, then there are no additional faxes to process, and no pending arrival of a fax. This condition will only hold after 4 PM and once all remaining faxes have been entered.

Also notice the event `changeStaff`, which is scheduled to occur at noon (240 minutes). Here we use the `setUnits` method of the `Resource` to change the staffing levels. The reader should look at the JavaSim `Resource` class and convince themselves that even if we reduce the number of staff at noon the faxes in process will not be affected.

The staffing policy to be simulated is set in the `myInit` method of `FaxCenter`.

The method `nsppFax` generates the interarrival times for faxes with the desired time-varying rate; this method will be described in Chap. 6.

```
/**
 * Run FaxCenter simulation
 */

private void runSimulation() {

  EventNotice nextEvent;

  for(int reps = 0; reps < 10; reps++) {
    javaSim.javaSimInit();
    agents.setUnits(numAgents);
    specialists.setUnits(numSpecialists);
    javaSim.schedule
      ("Arrival", nsppFax(aRate, maxRate, nPeriods, period, 0));
    javaSim.schedule("ChangeStaff", 4.0 * 60.0);
    do {
      nextEvent = javaSim.calendarRemove();
      javaSim.setClock(nextEvent.getEventTime());
      if (nextEvent.getEventType() == "Arrival") {
        arrival();
      } else if (nextEvent.getEventType() == "EndOfEntry") {
        endOfEntry((Entity) nextEvent.getWhichObject());
      } else if (nextEvent.getEventType() == "EndOfEntrySpecial") {
        endOfEntrySpecial((Entity) nextEvent.getWhichObject());
      } else if (nextEvent.getEventType() == "ChangeStaff") {
        agents.setUnits(numAgentsPM);
        specialists.setUnits(numSpecialistsPM);
      }

    } while(javaSim.calendarN() > 0); // stop when event calendar
    // empty

    javaSim.report(regularWait.mean(), reps + 1, 0);
    javaSim.report(regularQ.mean(javaSim.getClock()), reps + 1, 1);
    javaSim.report(agents.mean(javaSim.getClock()), reps + 1, 2);
    javaSim.report(specialWait.mean(), reps + 1, 3);
    javaSim.report(specialQ.mean(javaSim.getClock()), reps + 1, 4);
    javaSim.report(specialists.mean(javaSim.getClock()), reps + 1, 5);
    javaSim.report(regular10.mean(), reps + 1, 6);
    javaSim.report(special10.mean(), reps + 1, 7);
    javaSim.report(javaSim.getClock(), reps + 1, 8);
  }
}
```

Fig. 4.20 Main program for service center simulation.

Figure 4.21 contains the arrival and end-of-entry events for faxes at the Entry Agents. The next arrival is scheduled only if `javaSim.getClock() < runLength`; in this way we cut off fax arrivals after 4 PM. The `endOfEntry` event passes 20% of the faxes directly and immediately to the Specialists by executing `specialArrival(departingFax)` and passing the `departingFax` entity. Equivalently, we could have scheduled a `specialArrival` event to occur zero time units into the future (or nonzero time units if it takes time to transport the fax).

The `record` method of the `DTStat regular10` is used to collect a 0 or 1 depending on whether the total wait was less than 10 minutes.

The arrival and end-of-entry events for the Specialists, shown in Fig. 4.22, work similarly to those of the Entry Agents.

Initializations that occur once are shown in Fig. 4.23.

Ten replications of this simulation with a staffing policy of 15 Entry Agents in the morning and 9 in the afternoon, and 6 Specialists in the morning and 3 in the afternoon, gives 0.98 ± 0.02 for the fraction of regular faxes entered in 10 minutes or less, and 0.81 ± 0.06 for the special faxes. The "$\pm$" are 95% confidence intervals. This policy appears to be close to the requirements, although if we absolutely insist on 80% for the special faxes then additional replications are needed to narrow the confidence interval.

```
  private void arrival() {
    // Schedule next fax arrival if < 4 PM
    if (javaSim.getClock() < runLength) {
      javaSim.schedule
      ("Arrival", nsppFax(aRate, maxRate, nPeriods, period, 0));
    } else {
      return;
    }

    // Process the newly arriving Fax

    Entity fax = new Entity(javaSim.getClock());
    if (agents.getBusy() < agents.getUnits()) {
      agents.seize(1, javaSim.getClock());
      javaSim.schedulePlus("EndOfEntry",
        generator.normal(meanRegular, varRegular, 1), fax);
    } else {
      regularQ.add(fax, javaSim.getClock());
    }
  }

  private void endOfEntry(Entity departingFax) {
    double wait;

    // record wait time of regular; move in if special

    if (generator.uniform(0, 1, 2) < 0.2) {
      specialArrival(departingFax);
    } else {
      wait = javaSim.getClock() - departingFax.getCreateTime();
      regularWait.record(wait);
      if (wait < 10) {
        regular10.record(1);
      } else {
        regular10.record(0);
      }
    }

    // Check to see if there is another Fax; if yes start entry
    // otherwise free the agent

    if (regularQ.numQueue() > 0
      && agents.getUnits() >= agents.getBusy()) {
      departingFax = (Entity) regularQ.remove(javaSim.getClock());
      javaSim.schedulePlus
        ("EndOfEntry", generator.normal(meanRegular, varRegular, 1), departingFax);
    } else {
      agents.free(1, javaSim.getClock());
    }
  }
```

Fig. 4.21 Events for Entry Agents.

```
  private void specialArrival(Entity specialFax) {
    // if special agent available, start entry by seizing the special agent

    if (specialists.getBusy() < specialists.getUnits()) {
      specialists.seize(1, javaSim.getClock());
      javaSim.schedulePlus
        ("EndOfEntrySpecial", generator.normal(meanSpecial, varSpecial, 3), specialFax);
    } else {
      specialQ.add(specialFax, javaSim.getClock());
    }
  }

  private void endOfEntrySpecial(Entity departingFax) {
    double wait;

    // record wait time and indicator if < 10 minutes
    wait = javaSim.getClock() - departingFax.getCreateTime();
    specialWait.record(wait);
    if (wait < 10) {
      special10.record(1);
    } else {
      special10.record(0);
    }

    // check to see if there is another Fax; if yes start entry
    // otherwise free the specialist

    if (specialQ.numQueue() > 0
      && specialists.getUnits() >= specialists.getBusy()) {
      departingFax = (Entity) specialQ.remove(javaSim.getClock());
      javaSim.schedulePlus
        ("EndOfEntrySpecial", generator.normal(meanSpecial, varSpecial, 3), departingFax);
    } else {
      specialists.free(1, javaSim.getClock());
    }

  }
```

Fig. 4.22 Events for Specialists.

4.6.1 Issues and Extensions

1. There are many similarities between the programming for this simulation and the event-based simulation of the $M/G/1$ queue. However, there is an important difference that is due to having multiple agents. For the $M/G/1$ queue, a single `FIFOQueue` object held both the customer in service (who is at the front of the queue) and the customers waiting for service. This approach does not work for the Fax Center because when there are multiple agents the faxes need not complete entry in the same order in which they arrive. To accommodate this, the `FIFOQueue` holds only those faxes waiting for entry, and the `Entity` representing a fax that is being entered is stored with the `EventNotice` for the end-of-entry event. This is accomplished by the statement

   ```
   javaSim.schedulePlus
     ("EndOfEntry",generator.normal(meanRegular, varRegular, 1), fax);
   ```

   `schedulePlus` allows an object (`departingFax` in this case) to be assigned to the `.whichObject` field of the `EventNotice`. The `Entity` can then be passed to the event using the statement

   ```
   endOfEntry((Entity) nextEvent.getWhichObject());
   ```

2. The fax entry times were modeled as being normally distributed. However, the normal distribution admits negative values, which certainly does not make sense. What should be done about this? Consider mapping negative values to 0, or generating a new value whenever a negative value occurs. Which is more likely to be realistic and why?

Exercises

1. For the hospital problem, simulate the current system in which the receptionist's service time is well modeled as having an Erlang-4 distribution with mean 0.6 minutes. Compare the waiting time to the proposed electronic kiosk alternative.
2. Simulate an $M(t)/G/\infty$ queue where G corresponds to an Erlang distribution with fixed mean but try different numbers of phases. That is, keep the mean service time fixed but change the variability. Is the expected number if queue sensitive to the variance in the service time?
3. Modify the SAN simulation to allow each activity to have a different mean time to complete (currently they all have mean time 1). Use a Java collection (or one of its an implementing classes, such as ArrayList) to hold these mean times.
4. Try the following numbers of steps for approximating the value of the Asian option to see how sensitive the value is to the step size: $m = 8, 16, 32, 64, 128$.
5. In the simulation of the Asian option, the sample mean of 10,000 replications was 2.198270479, and the standard deviation was 4.770393202. Approximately

```
private void myInit() {
  String simulationName = "Fax Center";
  javaSim = new JavaSim(simulationName);
  // initialize the random number generator
  generator = new Rng();
  meanRegular = 2.5;
  varRegular = 1.0;
  meanSpecial = 4.0;
  varSpecial = 1.0;
  runLength = 480.0;
  numAgents = 15;
  numAgentsPM = 9;
  numSpecialists = 6;
  numSpecialistsPM = 3;
  // Add queues, resources and statistics that need to be
  // initialized between replications to the global collections
  regularWait = new DTStat();
  specialWait = new DTStat();
  regular10 = new DTStat();
  special10 = new DTStat();
  regularQ = new FIFOQueue(javaSim);
  specialQ = new FIFOQueue(javaSim);
  agents = new Resource(javaSim);
  specialists = new Resource(javaSim);
  javaSim.addDTStat(regularWait);
  javaSim.addDTStat(specialWait);
  javaSim.addDTStat(regular10);
  javaSim.addDTStat(special10);
  javaSim.addQueue(regularQ);
  javaSim.addQueue(specialQ);
  javaSim.addResource(agents);
  javaSim.addResource(specialists);
  javaSim.report("Ave Reg Wait", 0, 0);
  javaSim.report("Ave Num Reg Q", 0, 1);
  javaSim.report("Agents Busy", 0, 2);
  javaSim.report("Ave Spec Wait", 0, 3);
  javaSim.report("Ave Num Spec Q", 0, 4);
  javaSim.report("Specialists Busy", 0, 5);
  javaSim.report("Reg < 10", 0, 6);
  javaSim.report("Spec < 10", 0, 7);
  javaSim.report("End Time", 0, 8);
  // Arrival process data
  nPeriods = 8;
  period = 60.0;
  maxRate = 6.24;
  aRate = new double[8];
  aRate[0] = 4.37;
  aRate[1] = 6.24;
  aRate[2] = 5.29;
  aRate[3] = 2.97;
  aRate[4] = 2.03;
  aRate[5] = 2.79;
  aRate[6] = 2.36;
  aRate[7] = 1.04;
}
```

Fig. 4.23 Initializations for service center simulation.

how many replications would it take to decrease the relative error to less than 1%?

6. For the service center, increase the number of replications until you can be confident that that suggested policy does or does not achieve the 80% entry in less than 10 minutes requirement for special faxes.
7. For the service center, find the minimum staffing policy (in terms of total number of staff) that achieves the service-level requirement. Examine the other statistics generated by the simulation to make sure you are satisfied with this policy.
8. For the service center, suppose that Specialists earn twice as much as Entry Agents. Find the minimum cost staffing policy that achieves the service-level requirement. Examine the other statistics generated by the simulation to make sure you are satisfied with this policy.
9. For the service center, suppose that the staffing level can change hourly, but once an Agent or Specialist comes on duty they must work for four hours. Find the minimum staffing policy (in terms of total number of staff) that achieves the service-level requirement.
10. For the service center, pick a staffing policy that fails to achieve the service level requirements by 20% or more. Rerun the simulation with a replication being defined as exactly 8 hours, but do not carry waiting faxes over to the next day. How much do the statistics differ using the two different ways to end a replication?
11. The method `nsppFax` is listed below. This method implements the thinning method described in Sect. 4.2 for a nonstationary Poisson process with piecewise-constant rate function. Study it and describe how it works.

```
/**
 * This function generates interarrival times from a NSPP
 * with piecewise constant arrival rate over a fixed time
 * of Period*NPeriod time units
 *
 * @param aRate
 *          array of arrival rates over a common length Period
 * @param maxRate
 *          maximum value of ARate
 * @param nPeriods
 *          number of time periods in ARate
 * @param period
 *          time units between (possible) changes in arrival rate
 * @param stream
 *          seed for random number generator
 * @return
 *          an interarrival time from a non-stationary Poisson process
 */

private double nsppFax(double[] aRate, double maxRate,
  int nPeriods, double period, int stream) {
  double possibleArrival = javaSim.getClock() +
  generator.expon(1.0 / maxRate, stream);
  int i = Math.min(nPeriods,
  (int) Math.ceil(possibleArrival / period));
```

```
    while(generator.uniform(0, 1, stream) >=
      aRate[i - 1] / maxRate) {
      possibleArrival += generator.expon(1.0 / maxRate, stream);
      i = Math.min(nPeriods, (int) Math.ceil(possibleArrival / period));
    }

    return possibleArrival - javaSim.getClock();
  }
```

12. Beginning with the event-based $M/G/1$ simulation, implement the changes necessary to make it an $M/G/s$ simulation (a single queue with any number of servers). Keeping $\lambda = 1$ and $\tau/s = 0.8$, simulate $s = 1,2,3$ servers and compare the results. What you are doing is comparing queues with the same service capacity, but with 1 fast server as compared to two or more slower servers. State clearly what you observe.
13. Modify the Java event-based simulation of the $M/G/1$ queue to simulate an M/G/1/c retrial queue. This means that customers who arrive to find c customers in the system (including the customer in service) leave immediately, but arrive again after an exponentially distributed amount of time with mean `meanTR`. Hint: The existence of retrial customers should not affect the arrival process for first-time arrivals.
14. This problem assumes a more advanced background in stochastic processes. In the simulation of the $M(t)/M/\infty$ queue there could be a very large number of events on the event calendar: one "Arrival" and one "Departure" for *each* car currently in the garage. However, properties of the exponential distribution can reduce this to no more than two events. Let $\beta = 1/\tau$ be the departure rate for a car (recall that τ is the mean parking time). If at any time we observe that there are N car in the garage (no matter how long they have been there), then the time until the first of these cars departs is exponentially distributed with mean $1/(N\beta)$. Use this insight to build an $M(t)/M/\infty$ simulation with at most two pending events, next arrival and next departure. Hint: Whenever an arrival occurs the distribution of the time until the next departure changes, so the scheduled next departure time must again be generated.
15. The phone desk for a small office is staffed from 8 AM to 4 PM by a single operator. Calls arrive according to a Poisson process with rate 6 per hour, and the time to serve a call is uniformly distributed between 5 and 12 minutes. Callers who find the operator busy are placed on hold, if there is space available, otherwise they receive a busy signal and the call is considered "lost." In addition, 10% of callers who do not immediately get the operator decide to hang up rather than go on hold; they are not considered lost, since it was their choice. Because the hold queue occupies resources, the company would like to know the smallest capacity (number of callers) for the hold queue that keeps the daily fraction of lost calls under 5%. In addition, they would like to know the long-run utilization of the operator to make sure he or she will not be too busy. Use JavaSim to simulate this system and find the required capacity for the hold queue. Model the callers as class `Entity`, the hold queue as class `FIFOQueue` and the operator as class `Resource`. Use the `Rng` methods `expon` and `uniform` for random-variate

generation. Use class `DTStat` to estimate the fraction of calls lost (record a 0 for calls not lost, a 1 for those that are lost so that the sample mean is the fraction lost). Use the statistics collected by class `Resource` to estimate the utilization.

16. Software Made Personal (SMP) customizes software products in two areas: financial tracking and contact management. They currently have a customer support call center that handles technical questions for owners of their software from the hours of 8 AM to 4 PM Eastern Time.

    When a customer calls they the first listen to a recording that asks them to select among the product lines; historically 59% are financial products and 41% contact management products. The number of customers who can be connected (talking to an agent or on hold) at any one time is essentially unlimited. Each product line has its own agents. If an appropriate agent is available then the call is immediately routed to the agent; if an appropriate agent is not available, then the caller is placed in a hold queue (and listens to a combination of music and ads). SMP has observed that hang ups very rarely happen.

    SMP is hoping to reduce the total number of agents they need by cross-training agents so that they can answer calls for any product line. Since the agents will not be experts across all products, this is expected to increase the time to process a call by about 5%. The question that SMP has asked you to answer is how many cross-trained agents are needed to provide service at the same level as the current system.

    Incoming calls can be modeled as a Poisson arrival process with a rate of 60 per hour. The mean time required for an agent to answer a question is 5 minutes, with the actual time being Erlang-2 for financial calls, and Erlang-3 for contact management calls. The current assignment of agents is 4 for financial and 3 for contact management. Simulate the system to find out how many agents are needed to deliver the same level of service in the cross-trained system as in the current system.