An Object-Oriented Version of SIMLIB (a Simple Simulation Package)

Brian J. Huffman
School of Business
University of Wisconsin - River Falls
315 North Hall, River Falls, WI 54022-5001, USA

brian.j.huffman@uwrf.edu

Abstract

This paper introduces an object-oriented version of SIMLIB (an easy-to-understand discrete-event simulation package). The object-oriented version is preferable to the original procedural language versions of SIMLIB in that it is easier to understand and teach simulation from an object point of view. A single-server queue simulation is demonstrated using the object-oriented SIMLIB.

---

1. Introduction

Law and Kelton (2000) presents an easy-to-understand discrete-event simulation package called SIMLIB which "takes care of some standard list-processing tasks as well as several other common simulation chores, such as processing the event list, accumulating statistics, generating random numbers and observations from a few distributions, and writing out results."

A review of Law and Kelton (2000) on Amazon.com noted that an object-oriented version of SIMLIB could be written, and implied that and object-oriented version is needed. Law and Kelton (2000) mentions the growing popularity of the object-orientation in simulation and lists several advantages to the object-orientation (those advantages and others are mentioned in the next section).

The Java package introduced in this article is the first object-oriented version of SIMLIB. Porting SIMLIB from a procedural or third generation language like FORTRAN to an object-oriented language like Java is not so much a translation as a redesign. The result is a simulation package that is even easier to understand and use than the original SIMLIB (again, for reasons that will be given below).

The next section states the advantages of the new object-oriented simulation package compared to the old procedural language version. After that it will be shown how the Java files work to support discrete-event simulation. Finally a single-server queue simulation is implemented using the Java package.

2. The Advantages of an Object-Oriented Simulation Package

The object-oriented version of SIMLIB has many advantages. First, it has the same major advantage that any version of SIMLIB has: students using SIMLIB learn the basics of discrete-event simulation by being able to look "under the hood" of a simulation package. Students learn far more beginning with SIMLIB then they would by beginning with sophisticated commercial-quality simulation packages in which important details of the simulation implementation are hidden (and thus never understood). The student with SIMLIB experience is a more confident and competent user of commercial-quality simulation packages. In fact, the SIMLIB experience imparts the sort of deep knowledge of simulation without which the author of this paper could not have produced this work.

Second, in the object-oriented version of SIMLIB the relationship between the simulation model and the real world is more obvious than in the procedural language version (and therefore easier to teach and to understand). It is certainly easier to understand that a customer gets into a queue at the back then it is to understand that the arrival time for a customer is stored in list 3 (a queue) in order based on the data in position 2 (arrival time).

Hill (1996) and Meyer (1988) both commented on object-orientation as the apparent choice in discrete-event simulation. Meyer noted that object-orientation was probably used more in simulation than in any other domain. Hill explained that popularity by noting that the object-oriented

approach to the development of *any* software is analogous to writing a discrete-event simulation. In other words, one writes a discrete-event simulation when writing *any* object-oriented program.

Third, although Java is related to C and C++ (those who can read C or C++ can read Java), Java is easier to work with for two reasons. Java does not use pointers (which are difficult to teach or understand). Also, Java performs automatic garbage collection. That is, all objects that are no longer used in a program are automatically discarded saving computer memory space (and obviating the need to teach students to return memory occupied by unused objects to the system). Because it is not necessary to explicitly return memory to the system (as one must in C and C++), "memory leaks that are common in other languages like C and C++ cannot happen in Java" (Deitel 1998).

Fourth, Java (as an object-oriented language) supports data hiding so students cannot accidentally change simulation variables such as the clock time (which is a global variable in the original version of SIMLIB). Fifth, the Java version of SIMLIB allows the use of descriptive names where meaningless names and indexes are needed in the original version of SIMLIB. For example, the original SIMLIB had one master list of lists so the student had to know the index of the event list or the index of the list representing a customer queue. In the Java version a list can be given any name the student likes so the event list can be called something easily remembered like *eventList*, and the customer queue can be called something easily remembered like *queue* (the italicized names are used in the single server simulation which follows).

Sixth, since Java is the language of the web, the 7 files presented here could be rewritten as an applet and uploaded to run on the web. Finally, free versions of the Java compiler can be downloaded from Sun Microsystems at: http://java.sun.com/products/jdk/1.1/download-jdk-windows.html

Each student can have the same free Java compiler running on their own machine thus reducing the unnecessary use of computer labs.

3. How the Java Files Support Discrete-Event Simulation

This section will explain how the Java files work to support discrete-event simulation. It is not necessary for the reader to be familiar with either SIMLIB in general or the sequential language versions of SIMLIB in particular to be able to understand this section, but references to sequential language versions of SIMLIB will be made for the benefit of those who are familiar with it.

The sequential language versions of SIMLIB consist of 14 files: INITLK, FILE, REMOVE, TIMING, CANCEL, SAMPST, TIMEST, FILEST, OUTSAM, OUTTIM, OUTFIL, EXPON, IRANDI, UNIFRM. This object-oriented version consists of 5 main files and 2 support files. All 7 files are listed in the Appendix I. The 5 main files are: List.java, Timer.java, Random.java, ContinStat.java, and DiscreteStat.java. The 2 support files are: EmptyListException.java and SimObject.java. None of the 7 files does the work that was done in the CANCEL, FILEST, OUTSAM, OUTTIM, and OUTFIL files in the sequential language versions of SIMLIB; those files are useful, but unnecessary. Those interested in looking at the old sequential language version of SIMLIB can download the complete source code (as well as the code for four example simulations) from: http://www.mhhe.com/lawkelton.

3.1 Discrete-Event Simulation in General

Discrete-event simulations such as the single server simulation shown in the flowchart in Figure 1 proceed in three stages:

Stage 1. Variables and lists are initialized (the 2 boxes on the top of the flowchart)

Stage 2. A loop operates until a stopping condition is reached.

Stage 3. The results are printed out.

Every simulation needs several lists. A list might represent a queue at a bank, or jobs waiting to be processed on a machine. One special list, the event list, contains events in the order in which those events are to take place.

Every simulation needs a clock to track simulated time. In the second stage of the simulation the event list determines what happens and when it happens. Events are removed from the event list in chronological order. Just before an event is processed, the simulation clock is advanced to the time that the event takes place.

Every simulation needs methods for generating random numbers. For example, the simulation may need to generate a random interarrival time (the time between arrivals) to determine when the next customer will arrive. The designer of the simulation may want the interarrival time for customers to be according to an exponential distribution with a mean of 5 minutes, and the amount of time it takes to service a customer to be according to another distribution with a different mean.

Finally every simulation needs to track and record both continuous and discrete statistics. Continuous statistics are those which can be describe as a *time average*; that is, time is the denominator when averaging takes place. The time average number of customers in a queue is a continuous statistic. The utilization of a server is also a continuous statistic since utilization represents the how busy the server is *over time*. In contrast, those discrete statistics that are averages are obtained by dividing by something discrete or countable. The average time that customers wait in a queue is a discrete statistic since the average of customer waits is obtained by dividing the sum of all customer waits by the *number* of customers who had to wait. Not all discrete statistics are averages; the number of customer waits, for example, is a discrete statistic.

The java files presented here handle each of the needs just mentioned. List.java provides lists and methods for processing them. Timer.java provides the simulation clock. Random.java provides random numbers. Finally ContinStat.java and DiscreteStat.java can be used to track and record continuous and discrete statistics respectively. These files are described in detail in Section 3.3.

List.java (supported by EmptyListException.java and SimObject.java) does the same list processing that was done by the SIMLIB routines INITLK, FILE, REMOVE, and TIMING. The work formerly done by INITLK is an especially interesting case; that file was responsible for initialization, but initialization is unnecessary in Java since instances of primitive data types (built-in types like integer and floating point) have known initial values and objects (instances of user defined data types or "classes") are automatically initialized by special routines called constructors. More will be said about primitive data types, objects, and constructors in the next section.

Timer.java is the simulation clock. Simulation time was a real global variable in SIMLIB; it was supposed to be changed by the TIMING routine. However, since any routine has access to a global variable, the simulation clock could have been unintentionally changed by any routine. Simulation time in this object-oriented version of SIMLIB is maintained by a private floating point variable that belongs to the simulation clock. The simulation time cannot be accessed directly by any method (as functions are called in Java) outside the Timer class so it cannot be unintentionally changed.

Random.java generates random numbers doing the work that was done by EXPON, IRANDI, and UNIFRM. ContinStat.java and DiscreteStat.java do what TIMEST and SAMPST did respectively.

3.2 The Java Language

This section will provide some of the additional knowledge that one who is already familiar with C or C++ would need to understand the Java SIMLIB package. Java is a C derivative and most of the meaning of Java code will be clear to anyone familiar with either C or C++.

Java has primitive types such as byte, int, short, and long (all integers); float, and double (floating-point numbers); char (characters); and boolean. Java also allows users to establish their own types or "classes." Each of the files in the Java SIMLIB package defines a type. For example, the file Timer.java defines a timer type so users can declare a timer variable just as they can an integer variable. Also, just as an integer that will be used as a counter could be declared as:

```
int counter;
```


so a Timer that is to be used as a clock could be declared as:

```
Timer clock;
```

Just as an integer could be simultaneously declared and instantiated (initialized) as:

```
int counter = 0;
```

so a Timer can be simultaneously declared and instantiated as:

```
Timer clock = new Timer();
```

As noted in the last section, functions in Java are called "methods." There are three types of method calls in the Java SIMLIB package. The first type of method call is a call to create an object. That type of method is called is called a "constructor." Constructors always have the same name as the class. The constructor call "instantiates" (declares and initializes) an object. The above line of code was an example of the object "`clock`" being instantiated. Recall that clock is of class "`Timer`" so the constructor is also named "`Timer`."

Objects do not exist until they are instantiated. The line:

```
Timer clock;
```

merely declares the intention to create an object called "`clock`", but does not instantiate the object since the constructor was not called.

The second type of method call used in the SIMLIB package is a call made by an object to a method that belongs to that object. For example, "`Timer`" objects such as "`clock`" own certain methods. Timers can call a method "`setTime`" to set the present time or "`getTime`" to retrieve the present time. Method calls consist of the object name followed by the dot operator followed by the method name as shown in two examples here:

```
clock.setTime(0);
//Sets the present time as zero.

clock.getTime();
//Retrieves the present time.
```

The third and final type of method call is a call that does not involve an object at all. The file Random.java is different from Timer.java, for example, in that users may not create "Random" type objects (there is no "Random" constructor in Random.java). Instead the Random.java file consists of a number of methods that can be called to generate various sorts of random numbers. This type of method call consists of the class name followed by the dot operator followed by the method name. The example here returns an exponentially distributed random number with a mean of 8 using random number stream number 2:

```
Random.expon(8, 2);
```

Note that the word "`Random`" in the call begins with a capital "`R`" while the word "`clock`" in the two calls illustrated before the last paragraph begins with a lower-case "`c`." It is a convention in Java to begin class names with upper case letters and object names with lower case letters. The convention is followed in this paper so it should be easy to distinguish between the second and third type of method calls.

3.3 The Java SIMLIB Package

List Processing – SimObject.java

Two of the Java SIMLIB files support list processing. They are: List.java and SimObject.java. If a list is like a railroad train, and each link is like a railroad car, then a SimObject is like a container that goes into a railroad car.

The SimObject.java file is the only file in the SIMLIB package that is only intended to serve as a prototype. That is, the user is supposed to customize the SimObject file, deciding what data a SimObject needs to carry and to write methods to set and retrieve (get) that data. The list itself (the "railroad train") and the nodes that contain the SimObject (the "railroad cars") and the methods for working with both of them will not be affected by any alterations the user makes to the SimObject file.

The user's version of the SimObject.java file would probably have variables like the "name" and "time" variables in the prototype version (since every simulation has an event list and each event object in that list would have a name and a time at which that

event is to take place). There would also need to be a corresponding set of set/get methods to set/get the event name and event time values. The next two lines show how the set/get methods work in the prototype version of SimObject; in this case the event object is an arrival of a customer at 200 minutes:

```
event.setName("arrival");

event.setTime(200);
```

The prototype SimObject.java file also has data items named: `arriveTime`, `jobType`, `taskNumber`, `tellerNo`, and `remainTime`. Each of those data items has its own corresponding set/get routines. These data items were needed for the single-server example demonstrated in this paper and for the other three examples in Appendix III. The time-shared computer model needed the arriveTime and remainTime data items. The bank model needed arriveTime and tellerNo, and the job shop model needed arriveTime, jobType, and taskNumber.

List Processing – List.java

As stated above, List.java provides lists and methods for handling lists. This section will explain List.java's operation in terms of the methods it contains. The convention from here onward will be to show a method's *signature* followed by an explanation of that method. The signature of a method is what one sees in the first line of the method's code. The syntax for the signatures used here is (roughly): the word "`public`", followed by the return type (if any), followed by the method's name (in bold), followed by an argument list in parenthesis. The user does not have to be concerned with the meaning of the words "`public`", "`synchronized`", or "`static`" in the methods shown below. A return type of "`void`" means that nothing is returned.

`public` **`List`**`(String s)`
This method constructs a new list with name "`s`."

`public` **`List`**`()`
This method constructs a new list with a default name of "`list`."

`public synchronized void` **`insertInOrder`**
`(Object insertItem, float newNodeIndex)`
This method inserts the object "`insertItem`" in a list in order determined by the "`newNodeIndex`." For example, in the bank simulation an event is inserted in the event list in order of when that event is to take place using the following statement:

```
eventList.insertInOrder(event,
event.getTime());
```

`public synchronized void`
`insertAtFront``(Object insertItem)`
This method inserts the object "`insertItem`" at the front of a list. This method might be used to simulate a LIFO (Last-In-First-Out) list (often called a "stack").

`public synchronized void` **`insertAtFront`**
`(Object insertItem, float newNodeIndex)`
This method works the same way as the last one, but also allows the user to assign an index to the inserted item.

`public synchronized void` **`insertAtBack`**
`(Object insertItem)`
This method inserts the object "`insertItem`" at the back of a list. This method might be used to simulate a FIFO (First-In-First-Out) list or an ordinary queue.

`public synchronized Object`
`removeFromFront``()`
This method has the word "`Object`" where the previous methods have had the word "`void`." As the reader might guess, "`Object`" is the return type of this method (the previous methods had "`void`" as return type since they don't return anything). This method returns the object at the front of a list. The following line of code would remove the first event from the event list:

```
event = eventList.removeFromFront();
```

`public synchronized Object`
`removeFromBack``()`
This method returns the object at the back of a list.

`public Boolean` **`isEmpty`**`()`
This method returns a value of "`null`" if a list is empty. It is (obviously) used to see if a list is empty.

`public void` **`print()`**
This method is used to print out information on all of the nodes in a list.

There is another class (in addition to the List class) in the List.java file, the ListNode class, which will not be discussed since its operation is of no concern to the user. It is the "railroad car" in the previously mentioned analogy, and its methods are only for the Lists use.

Simulation Time – Timer.java

The Java SIMLIB file Timer.java supports simulation time by allowing the user to declare an object for tracking time. There are three methods in this file.

`public` **`Timer()`**
This method is a constructor and is used to declare a timer object. In all four of the examples in this article (the single-server and the three in Appendix III) that object was called the "`clock`." The "`clock`" object was always constructed (instantiated)in exactly the same way:

```
Timer clock = new Timer();
```

`public void` **`setTime`**`(float time)`
This method sets the present time to the value of the floating-point argument "`time`."

`public float` **`getTime`**`()`
This method returns the present time.

Examples involving setTime and getTime were already given in Section 3.2, the section on the Java language.

Random Number Generation - Random.java

Simulations need random numbers and those numbers are generated by random number generating methods in Random.java. As stated in the section on the Java language, these methods are unusual in that they are not called by objects. Some of the methods in Random.java are only there to service other Random.java methods (they are not called directly by users). This section will not explain those service methods.

One of the service methods that will not be explained here is rand. The rand method is "the" essential method since all of the other methods call it to generate the [0,1] random variable they need to generate their random numbers. The code for rand (as well as details concerning its operation) can be found in Law and Kelton (2000) where it is in turn attributed to Marse and Roberts (1983). The "`Random`" methods which will be explained are:

`public static float` **`expon`**`(float rmean, int istrm)`
This method returns an exponentially distributed floating point number generated from random number stream "`istrm`" and with a mean "`rmean`."For example, interarrival times in the single server queue simulation were based on a mean interarrival time of 1.0 minutes (MEANARR) and using a stream number 1 (ARRSTR); the method call used was:

```
Random.expon(MEANARR, ARRSTR);
```

`public static int` **`irandi`**`(int nvalue, float probd[], int istrm)`
This method returns an integer from 1 to "`nvalue`" according to the cumulative probability distribution "`probd[]`" using the random number stream "`istrm`." For example, if we want to generate the number 1 with 10% probability, 2 with 30% probability, and 3 with 60% probability then the cumulative probability distribution "`probd[]`" is an array with 0.10 in the first position, 0.10 + 0.30 = 0.40 in the second position, and 0.10 + 0.30 + 0.60 = 1.00 in the third position. The "`nvalue`" in this case would be 3, and "`istrm`" can be any integer (from 1 to 100) that the user chooses.

`public static int` **`randomInteger`**`(float probDistrib[], int stream)`
The method "`irandi`" will probably not be used since this method does the same thing and requires one less argument. Thus, "`probDistrib[]`" is a cumulative probability distribution, and "stream" is the number of the random number stream the user wants to use.

`public static float` **`unifrm`**`(float a, float b, int istrm)`
This method generates a random floating point

number uniformly distributed on the interval from "a" to "b" using the random number stream "istrm."

`public static float` **`erlang`**`(int m, float mean, int stream)`
This method generates an "m"-Erland random floating point number with a mean of "mean" using random number stream "stream." A 2-Erland distribution was used in the job shop model since job processing times were assumed to depend on two factors (those factors being the item being produced and the nature of the equipment used at the current stage in the production of the item in question).

Recording/Reporting Continuous Statistics - ContinStat.java

Simulations will generally involve recording and reporting some continuous statistics such as the time average number of customers in a queue or the utilization of a server (the average percentage of time the server is being used). The following methods are used to record and report these statistics:

`public` **`ContinStat`**`(float value, float present)`
This is a constructor. It creates a continuous statistic at the "present" time with the initial value of "value." This method could be used as shown here to create a continuous statistic for tracking the time average size of a queue (the initial queue size is "queueSize" and the time is 0):

```
aveQueueSize = new
ContinStat(queueSize, 0);
```

`public void` **`recordContin`**`(float value, float present)`

This method is called every time the value of a continuous statistic changes. It records the fact that at the "present" time the statistic took on a new value of "value." Continuing with the last code example, if another customer joined the queue, the proper recordContin method call would be:

```
aveQueueSize.recordContin(queueSize,
clock.getTime());
```

Note that the present time was obtained by using the value returned by the `getTime` procedure which was called by the simulation clock object.

`public float` **`getContinAve`**`(float present)`
This method is for reporting the time average of the continuous statistic at the "present" time. Ordinarily this method call would be done once the simulation was over.

`public float` **`getContinMax`**`()`
This method is for reporting the maximum value observed for the continuous statistic during the course of the simulation. Again, this method call would be ordinarily done once the simulation was over. It could be used, for example, to see how long a queue was at maximum.

`public float` **`getContinMin`**`()`
This method is for reporting the minimum value observed for a continuous statistic during the course of the simulation.

Recording/Reporting Discrete Statistics - DiscreteStat.java

Simulations will also generally involve recording and reporting discrete statistics such as the average length of time a customer has to wait in queue. The following methods are used to record and report these statistics:

`public` **`DiscreteStat`**`()`
This is a constructor. It creates a discrete statistic. Unlike the ContinStat constructor no time or initial value arguments are necessary.

`public void` **`recordDiscrete`**`(float value)`
This method is called every time the value of a discrete statistic changes. It records the fact that the statistic took on a new value of "value." It does not need to know the present time (as does its continuous statistic counterpart).

The next 5 methods are all for reporting about the discrete statistics and would all be called once the simulation has ended:

`public float` **`getDiscreteSum()`**
This method returns the sum of the observed values of the discrete statistic.

`public int` **`getDiscreteObs()`**
This method returns the number of observations made (the number of times the `recordDiscrete` method was called).

`public float` **`getDiscreteMax()`**
This method returns the maximum value which the discrete statistic took on during the simulation.

`public float` **`getDiscreteMin()`**
This method returns the minimum value which the discrete statistic took on during the simulation.

`public float` **`getDiscreteAverage()`**
This method returns the average of the values which the discrete statistic took on during the simulation. It is the same as the value returned by `getDiscreteSum` divided by the value returned by `getDiscreteObs`.

The Seventh (and Last) File - EmptyListException.java

The last file contains only one method, a constructor named `EmptyListException`, the only purpose of which is to warn the user if the simulation attempts to remove an object from an empty list. This method is not called by the user so it will not be explained.

4. A Single-Server Queue Simulation Implemented with the Java Package

This section provides an example of how the 7 Java files in the SIMLIB package can be used in a discrete-event simulation. The Java program demonstrated here, SingleServer.java, implements a single-server queueing system and its code can be found in the Appendix I along with the 7 Java files that make up the SIMLIB package.

The single-server example is one of four examples covered in Chapter 2 of Law and Kelton (2000). Instructions for loading and running these examples in Microsoft Visual J++ are included in Appendix II. The code for the remaining examples can be found in Appendix III. The other examples are not explained here (except by way of their internal documentation), but are explained in detail in Chapter 2 of Law and Kelton (2000).

In the single-server queueing system interarrival times (the difference in time between successive customer arrivals) are independent identically distributed (IID) random variables. If a customer arrives and the server is not busy the customer is served immediately, otherwise the customer joins the queue at the back and is served after all others in the queue have been served. The service time is also an IID random variable.

The simulation runs until 11 customers have been served. A continuous (or time-averaged) statistic and a discrete statistic are both collected during the simulation and are printed when the simulation run is completed. The continuous statistic is the average size of the queue. The discrete statistic is the average time customers spent waiting in the queue.

The explanation of the SingleServer.java program which follows will reference the code in that file. The following flowchart will also help to understand the program.

4.1 Initialization

The simulation begins by initializing several variables and instantiating (creating) several objects. First a `Timer` object called `clock` is instantiated. This object is the simulation clock, and it is set to zero by the statement: `clock.setTime(0)`.

Next the integer variable busy is set to zero indicating that the server is not busy (it is set to 1 to indicate a busy server). An integer variable `queueSize` is also set to zero indicating that there are no customers waiting in the queue.

Two list objects (two lists) `eventList` and `queue` are then instantiated. As mentioned above, the original versions of SIMLIB used one master list of lists requiring the student to remember which index corresponded to which list. In this object-oriented version of SIMLIB the student simply instantiates a

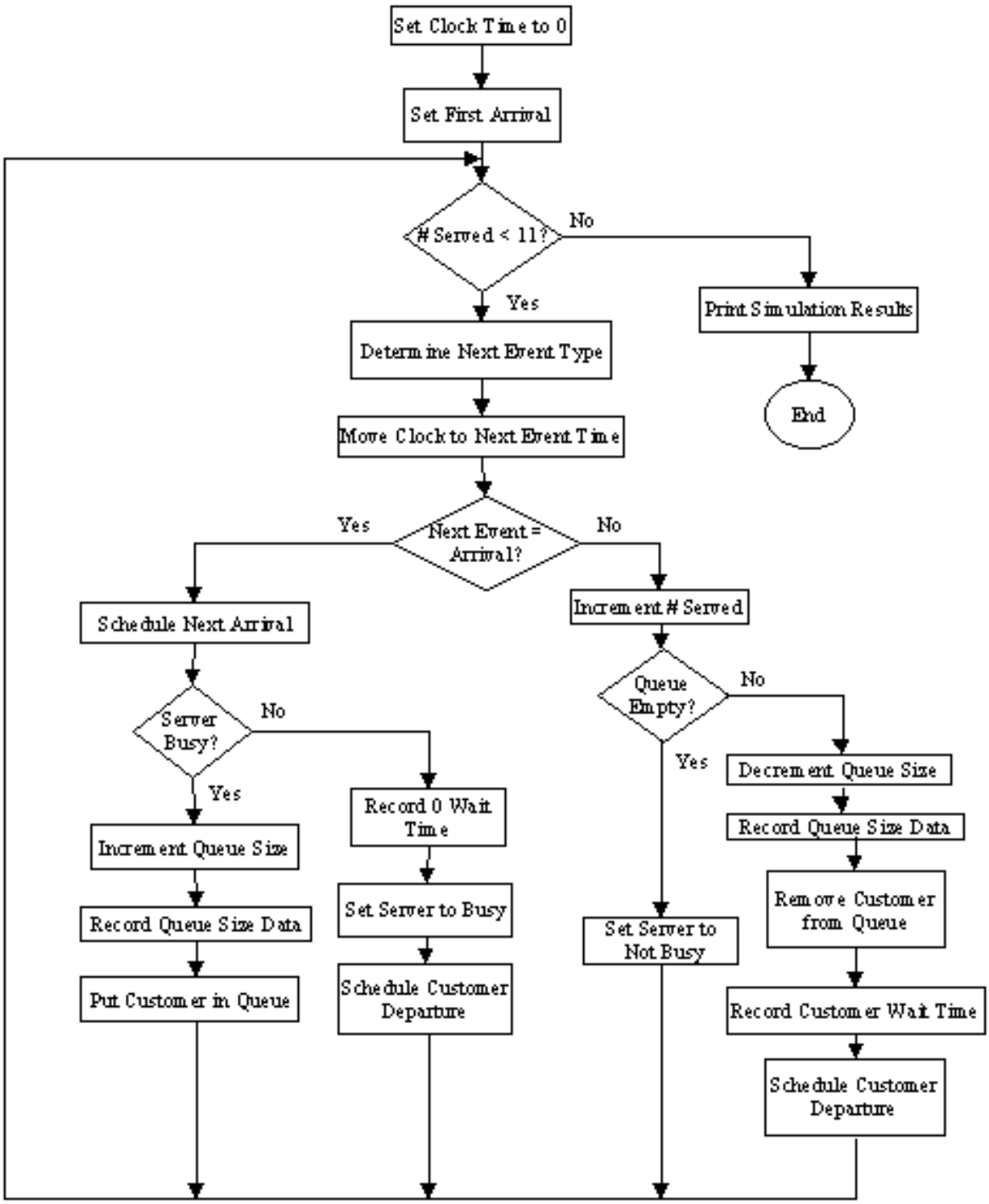

Figure 1: Flow Chart for the Single Server Simulation

new list whenever one is needed. Each of the lists can be given any meaningful name as is the case here; the list called `eventList` holds the events and the list called queue holds the queued customers.

Next a discrete statistic object called `waitTime` and a continuous statistic object called `sizeQueue` are instantiated. Once again the original versions of SIMLIB would not have allowed meaningful names to be used.

Four constants are declared next. They are the mean interarrival time, `MEANARR`; the mean processing time, `MEANPRO`; the random number stream to be used for interarrival times, `ARRSTR`; and the random number stream to be used for processing times, `PROSTR`. There are 100 random number streams so `ARRSTR` and `PROSTR` could be set to any number from 1 to 100.

In the next few lines of code four SimObjects are created. SimObjects are members of the lists `eventList` and `queue`. That is, SimObjects are either customers or events. The four SimObjects are `waitingCustomer`, a customer being added to the `queue`; `event`, an event being added to the eventList; `dequeued`, a customer who has just left the queue to be served; and `removed`, an event which has just been removed from the eventList. The program could have been written using only one SimObject which would have done the job of the four objects, but instantiating four objects (one object for each task) makes it easier to read and understand the code.

The event of the first customer arrival is set in the three statements beginning with `event.setName("arrive")` which sets the name of that event to "arrive." The second statement `event.setTime(clock.getTime() + Random.expon(MEANARR, ARRSTR))` sets the customer arrival time as the present clock time plus the interarrival time; as mentioned earlier the interarrival time is a random number with an exponential distribution. The interarrival time will have a mean `MEANARR` taken from stream `ARRSTR`. The third statement `eventList.insertInOrder (event, event.getTime())` inserts the event in the eventList in order according to when that event is to take place (that is what "`insertInOrder`" is meant to convey).

The next part of the simulation program is enclosed in "try" and "catch" blocks for exception handling. The exception would be a student programming error involving a list. An attempt to remove a customer or an event from an empty list would be an example of this type of error. All the student needs to know is that the initialization goes before the "try" and "catch" blocks, the bulk of the program goes in the "try" block, and the final output goes after the "catch" block.

The bulk of the single server program is in the "try" block within a "while" block which begins, `while(numServed < 11)`. That is, the simulation will run until 11 customers have been served. The first two statements in the while block:

```
removed =
(SimObject)eventList.removeFromFront();

clock.setTime(removed.getTime());
```

remove the next event from the `eventList` and move the clock forward to the time that the event takes place. As mentioned earlier, the removed event (the object removed from the `eventList`) is called `removed`.

The `removed` event, removed, is either the arrival of a customer wanting service or the departure of a customer who has been served. The event type is determined at the top of the "`if`" block by the statement:

```
if (removed.getName() == "arrive")
```

This statement evaluates to true if the name of the removed event is "`arrive`" (an arrival event). If an event is not an arrival then it is a departure (there are only those two types of events in the single server simulation). Arrivals are handled by the following "`if`" block and departures are handled by the following "`else`" block.

4.2 Arrival Event

If the event is an arrival, the next arrival is scheduled by the four statements:


```
event = new SimObject();

event.setName("arrive");

event.setTime(clock.getTime() +
Random.expon(MEANARR, ARRSTR));

eventList.insertInOrder(event,
event.getTime());
```

The first statement creates a new event. The function of next three statements is the same as when they were used to define the first arrival (explained above).

Next the arriving customer is either put into a waiting queue or served depending on whether or not the server is busy. The server's status is determined at the top of another "`if`" block by the statement:

```
if (busy == 1)
```

which evaluates to true if the server is busy. If the server is busy the `queueSize` is incremented:

```
queueSize++;
```

the continuous statistic `sizeQueue` is updated:

```
sizeQueue.recordContin((float)queueSize,
clock.getTime());
```

and a new `SimObject` object called `waitingCustomer` is instantiated and put in the queue:

```
waitingCustomer = new SimObject();

waitingCustomer.setTime(clock.getTime();

queue.insertAtBack(waitingCustomer);
```

If the server is not busy the "`else`" block executes. The server waits on the customer immediately. The fact that the newly arriving customer didn't have to wait is noted (a wait of zero is recorded):

```
waitTime.recordDiscrete((float) 0);
```

the server is set to busy:

```
busy = 1;
```

and a new `SimObject` object event, the departure of the customer who just entered service, is scheduled:

```
event = new SimObject();

event.setName("depart");

event.setTime(clock.getTime() +
Random.expon(MEANPRO, PROSTR));

eventList.insertInOrder(event,
event.getTime());
```

4.3 Departure Event

If the event is a customer departure, another customer has been served so the number of customers served is incremented:

```
numServed++;
```

and if the queue is empty the server is set to not busy:

```
if(queue.isEmpty())
    busy = 0;
```

If the queue isn't empty, the queue size is decremented,

```
queueSize--;
```

the queue size continuous statistic is updated,

```
sizeQueue.recordContin((float)queueSize,
clock.getTime());
```

and the customer is dequeued (the customer in the front of the queue enters service):

```
dequeued =
(SimObject)queue.removeFromFront();
```

The time that customer spent in the queue is determined (that time is represented by a floating point variable, `delay`), and the delay for that customer is used to update the `waitTime` discrete statistic:

```
float delay = clock.getTime() -
dequeued.getTime();

waitTime.recordDiscrete(delay);
```

Finally, the departure of the customer that just entered service is scheduled:

```
event = new SimObject();
```



```
event.setName("depart");

event.setTime(clock.getTime() +
Random.expon(MEANPRO, PROSTR);

eventList.insertInOrder(event,
event.getTime));
```

4.4 Output

Output begins after the catch block outputs any errors the student may have made using the eventList or queue:

```
Catch (EmptyListException e)
    {
    System.err.println("\n" +
    e.toString());
    }
```

The output consists of the sizeQueue continuous statistic and the waitTime discrete statistic. The continuous statistic is ready to be printed out when the simulation ends. Here a variable f2 is set to the average queue size:

```
float f2 =
sizeQueue.getContinAve(clock.getTime());
```

The discrete statistic is not ready to be printed out. The discrete statistic that was collected was the total time that customers spent waiting in the queue so it needs to be divided by the number of waits observed (DiscreteObs) before it can be printed out. Here the variable f1 is set to the total time that customers spent waiting in the queue:

```
float f1 = waitTime.getDiscreteSum();

int i1 = waitTime.getDiscreteObs();

f1 = f1/i1;
```

Finally, the discrete and continuous statistics are printed out:

```
System.out.println
("The average wait in queue was: " +
String.valueOf(f1));

System.out.println
("The average size of the queue was:"+
String.valueOf(f2));
```

References:

Deitel, H. M., and P. J. Deitel (1998), *Java How to Program* (2nd ed.), Prentice Hall, Upper Saddle River, NJ.

Hill, David R. C. (1996), *Object-Oriented Analysis and Simulation*, Addison-Wesley, Harlow, England.

Law, A. M. and W. D. Kelton (2000), *Simulation Modeling and Analysis* (3rd ed.), McGraw-Hill, New York, NY.

Marse, K. and S.D. Roberts (1983), "Implementing a Portable FORTRAN Uniform (0,1) Generator," *Simulation*, 41: pp.135-139.

Meyer, B. (1988), *Object Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ.

Appendix I:

Links to Source Files

The Java Simulation Main Files (5 Files)

List.java
http://ite.informs.org/vol2no1/Huffman/list.java

Timer.java
http://ite.informs.org/vol2no1/Huffman/Timer.java

Random.java
http://ite.informs.org/vol2no1/Huffman/Random.java

ContinStat.java
http://ite.informs.org/vol2no1/Huffman/ContinStat.java

DiscreteStat.java
http://ite.informs.org/vol2no1/Huffman/DiscreteStat.java

The Java Simulation Support Files (2 Files)

EmptyListException.java
http://ite.informs.org/vol2no1/Huffman/EmptyListException.java

SimObject.java
http://ite.informs.org/vol2no1/Huffman/SimObject.java

The Single Server Simulation (1 File)

SingleServer.java
http://ite.informs.org/vol2no1/Huffman/SingleServer.java

Appendix II:

Loading Files into Microsoft Visual J++

1 When you start Microsoft Visual J++ the "New Project" window opens automatically, if it isn't open it can be opened by clicking on "File" on the main menu and then selecting "New Project" (the top item on the drop-down menu).

2 In the "New Project" window you will see 3 tabs on top. Those tabs are "New", "Existing", and "Recent." Click on the "New" tab.

3 In the "Name:" textbox at the bottom of the "New Project" window you will see a suggested project name. Change it if you like. The name will be attached to the directory in which will be stored all of the files for the application you are building. The "Location:" textbox shows the path to that file.

4 Click the "Open" button on the bottom of the "New Project" window.

5 The "New Project" window closes and you should see a window called "Project Explorer – Project Name" (where Project Name is the name you assigned to the project in step 3). If the "Project Explorer" window isn't open click "View" on the main menu then click "Project Explorer" (which is the top item on the drop-down menu).

6 The "Add Item" window pops up. There are two tabs on this window: "New" and "Existing." Click on the "Existing" tab.

7 Navigate around on your hard drive by clicking on the folder icon to the right of the "Look in:" textbox or by clicking on the folder icons below that line until you find the java files you want to load (the files all have a ".java" extension).

8 Hold down the shift key on your keyboard and click on every ".java" file you want to load. This will be the main program file ("SingleServer.java" for example) and all of the java files that support the simulation (ContinStat.java, DiscreteStat.java, EmptyListException.java, List.java, Random.java, SimObject.java, and Timer.java).

9 Click the open button on the bottom of the "Add Item" window.

10 You can look at oy double clicking on its name in the "Project Explorer – Project Name" window.

11 To run the project. Click "Build" on the main menu and click "Build" on the drop-down menu. Then click "Debug" on the main menu and "Start" on the drop-down menu. Since this is the first time you are running the program a "Project Name Properties" window will pop up. Select the "Launch" tab. Make sure the "Default" radio button is selected and make sure the name of the main program file ("SingleServer.java" in this example) appears in the "When the project runs, load:" textbox. Click the "OK" button at the bottom of the window and the application will run.

12 When you leave Microsoft j++ a window will pop up asking you if you want to save two files: "Project Name.sln" and "Project Name.vjp." Selecting "Yes" will save all the files you have loaded.

Appendix III:

The Other Three Examples

The Time-Shared Computer Model

This model is equivalent to the one covered in Section 2.5 in Law and Kelton (2000). The file name is Time_Share.java. The following link is to the Time_Share.java source code which is a text file.

http://ite.informs.org/vol2no1/Huffman/Time_Share.java

The Bank Teller Model

This model is equivalent to the one covered in Section 2.6 in Law and Kelton (2000). The file name is Bank.java. The following link is to the Bank.java source code which is a text file.

http://ite.informs.org/vol2no1/Huffman/Bank.java

The Job Shop Model

This model is equivalent to the one covered in Section 2.7 in Law and Kelton (2000). The file name is Job_Shop.java. The following link is to the Job_Shop.java source code which is a text file.

http://ite.informs.org/vol2no1/Huffman/Job_Shop.java

