Java Simulation Library (JSL): An Open-Source Object-Oriented Library for Discrete-Event Simulation in Java

Manuel D. Rossetti, PhD PE*
Department of Industrial Engineering,
University of Arkansas, Fayetteville, AR 72703, USA
E-mail: rossetti@uark.edu
*Corresponding author

Accepted Pending Final Revisions

Abstract: This paper describes the design and functionality of an open-source object-oriented library for executing discrete-event simulation models in the Java programming language. The structure of the library is described in terms of its packages, class structure, and functionalities. The purpose of this paper is to provide an understanding of the library so that practitioners and researchers can better utilize the library for simulation modelling. In addition, the expository nature of this paper can help new users or students learn the basics of discrete-event simulation from the library's design. The capabilities of the library are illustrated through a number of code examples.

Keywords: object-oriented, discrete event simulation, library

Reference to this paper should be made as follows: Place appropriate reference here

Biographical notes: Place author biography here

1 INTRODUCTION

According to Booch et al. (1999), a software design framework is "an architectural pattern that provides an extensible template for applications within a domain." A framework provides a set of classes that can be extended via sub-classing or used directly to solve a particular problem within a particular domain. This paper discusses a software design framework for developing object-oriented simulations. The concepts in the framework have been implemented in open source software called the Java Simulation Library (JSL) under the GNU General Public License (www.gnu.org).

Numerous organizations and individuals have developed object-oriented simulation software tools. These tools have been implemented using languages such as C, C++, and Java. For example, Schwetman's (1986) CSIM++™ is an extensive set of classes for performing process and event based simulations in C++. Joines and Roberts' YANSL (1996) supports the development of C++ simulations using the network modelling view. Marzolla (2004) supports the process view in C++. Healy and Kilgore (1997) describe Silk™, a comprehensive commercial extension of the Java simulation language for developing process-based simulations. In addition, Borshchev et al. (2000) describe the beginning architecture for AnyLogic™, a comprehensive commercial product built on Java that does discrete-event, agent-based, and dynamic simulation.

The paper by Rossetti et al. (2000) describes an object-oriented framework that is the predecessor for the software discussed in this paper. Some other discrete event libraries implemented in Java include L'Ecuyer et al. (2002), Page and Kreutzer (2005), Jacobs and et al. (2002), Garrido (2001), Kacer (2001), and Sánchez (2006).

It is not the purpose of this paper to contrast the JSL with the plethora of other Java based simulation libraries; however, some similarities and differences are worth pointing out. Both SSJ from L'Ecuyer et al. (2002) and

Copyright © 200? Inderscience Enterprises Ltd.

DesmoJ from Page and Kreutzer (2005) describe software that has very similar overall capabilities as the software discussed in this paper. That is, the ability to model discrete-event systems in Java using the event and process world-views with full support for random variate generation and statistical output collection. DesmoJ has an experiment and modelling paradigm that is similar in concept to the JSL's. The statistical and random utilities for both of those libraries are on par with what is provided within the JSL.

The libraries PSIM-J by Garrido (2001), J-SIM by Kacer (2001), SSJ by L'Ecuyer et al. (2002) and DesmoJ from Page and Kreutzer (2005) all focus on the process interaction approach implemented on top of Java's thread mechanism. While the JSL originally had process interaction support based on threads, see Rossetti et al. (2000), that implementation has been depreciated due to problems with the use of Java's threads. Perhaps the final difference with respect to other libraries has been the emphasis on designing the JSL to be flexible and extendable under an open-source paradigm. For example, the reliance on observer-based collection of statistics is somewhat unique to the JSL and provides great flexibility. In addition, as will be noted in the following sections, the JSL can readily use different random number generators, different event scheduling calendars, different statistical collection techniques, etc., which facilitated research of these topics within a consistent platform.

The Java Simulation Library (JSL) has been used in a number of research and educational settings; however, its implementation has not been fully described in the literature except through its many applications. Examples of the use of the JSL include, Hobbs et al. (2006), which used it to model radio frequency tracking on an air base and Rossetti and Chan (2003), which developed a prototype framework for simulating supply chains. Building on that work, Rossetti and Thomas (2006) developed a framework to simulate supply chains involving spare parts in a multi-echelon, multi-indentured setting. The work in Rossetti, Miman et al. (2006) extends the JSL's capabilities to handle large-scale multi-echelon inventory systems with any type of inventory policy. Finally, the supply chain work was integrated with a transport layer for modelling transportation networks (e.g. full-truck-load networks) in Rossetti and Nangia (2007). Since the JSL is open source, these implementations are also freely available.

The purpose of this paper is to describe the JSL's design and to illustrate how to use the JSL for basic discrete-event modelling. This paper is primarily explicative in nature. The notion of simulation libraries is not new; however, much can still be learned by examining the design and implementation of such software systems. This is especially relevant in an educational and research environment. Students of simulation can develop a deeper understanding of how simulation technology works by studying such designs. This is one of the primary reasons for this paper and why the software is open-source.

Since the JSL is divided into a number of Java packages (random, statistic, modelling, calendar, observer, etc), the paper is organized around the capabilities of these packages. Then, through the use of some examples and a textbook problem, the use of the software is illustrated. Finally, the paper summarizes some ideas for future expansion and use of the JSL.

2 JSL PACKAGES

The general purpose simulation support within the JSL is organized into four major packages: utilities, calendar, modeling, and observers. While Java does not support the notion of sub-packages, the modeling, observer, and utilities packages can be conceptualized as having sub-packages. The utilities package contains the random and statistic packages for random number generation and statistical collection. The modelling package has packages for modelling queues, resources, etc., essentially elements that a user would find in a simulation model. The observer package is built off of Java's notion of the Observer/Observable pattern to facilitate statistical collection and reports, as well as writing spreadsheet and database output. This section briefly overviews each of the major packages and their sub-packages starting with the utilities package.

2.1 The Utilities Package

The utilities package consists of classes that support the JSL in a general manner. It consists of sub-packages for reporting, forecasting, numerical computation, sampling random variables, and statistical collection. The following subsections provide an overview of the random and statistic packages.

2.1.1 The Random Package

The classes and interfaces within the random package support the sampling of random variables within the JSL. Figure 1 illustrates a few of the key interfaces and classes within the random package. The ability to generate random numbers is represented by the `RngIfc` interface. This interface combines the generation of pseudo-random numbers (via the `randU01()` method within the `RandU01Ifc` interface) with stream control via the `RandomStreamIfc` interface. Stream control, see L'Ecuyer (2001), allows the user to restart a pseudo-random sequence, jump to sub-sequences, and produce antithetic variates. The `RngStream` class (based on L'Ecuyer (2001)) provides a concrete implementation of the `RngIfc` interface. The organization of these interfaces also facilitates the incorporation of pseudo-random number generators that do not easily allow for the manipulation of streams.

Because of the flexibility of the `RngIfc` interface users of the random package can also easily generate correlated pseudo-random numbers. The generation of correlated processes is discussed in Law (2007). For example, the `AR1CorrelatedRng` class uses the Normal-to-Anything

Accepted Pending Final Revisions

Transformation as discussed in Banks et al. (2006), Cario and Nelson (1996, 1998), and Biller and Nelson (2003). To use correlated random numbers in a simulation, the user can simply supply an instance of the `AR1CorrelatedRng` class wherever an instance of the `RngIfc` interface is required. Other correlation induction algorithms can be easily added to the library by implementing the `RngIfc`.

In stochastic simulations, users require the generation of many kinds of random quantities or values (e.g. time series, stochastic processes, etc). For example, the random package provides a class, `AR1Normal`, which implements the `RandomIfc` interface to generate values from an autoregressive order 1 normal process. The `RandomIfc` interface supports this generation via its implementation of the `GetValueIfc`, `ParametersIfc`, and `RandomStreamIfc` interfaces. A class that implements the `RandomIfc` interface has the ability to manipulate streams and through its `getValue()` method return a value that is random. The `ParametersIfc` interface permits a general mechanism for setting the parameters of whatever stochastic process is being utilized to implement the underlying randomness. The `RandomIfc` interface also provides a method, `getSample()`, which should return an array of doubles that are generated by the underlying random process.

Insert Figure 1 about here

The random package provides support for random variable sampling based on probability distributions through the `DistributionIfc` interface. Any probability distribution that works within the JSL must adhere to this interface. An abstract base class, `Distribution`, is provided to facilitate the development of concrete implementations of the `DistributionIfc` interface. The `Distribution` class takes in an instance of the `RngIfc` interface to use as its source of pseudo-random numbers.

The JSL currently implements the following continuous distributions: beta, gamma, uniform, lognormal, normal, exponential, triangular, loglogistic, and Weibull. Other distributions can be easily added. In addition, the random package includes a general method to sample from truncated distributions via the `TruncatedDistribution` class. Truncated distributions are discussed on page 432 of Law (2007). The beta and gamma distributions also include static methods for computing the beta function, the gamma function, and the logarithm of the gamma function.

From a class hierarchy perspective, the JSL does not distinguish between continuous and discrete distributions as is done in other libraries (e.g. SSJ). Sub-classing from the Distribution class also provides a number of discrete distributions: `Poisson`, `Constant`, `DUniform`, `DEmpirical`, `NegativeBinomial`, `Geometric`, `Bernoulli`, and `Binomial`. The `Binomial` class also provides a numerically stable method for computing binomial coefficients. The current implementation of these distributions utilizes the inverse transform method for generating random variates but overriding the methods can easily provide other algorithms.

The random package also provides support for shifted distributions, generating permutations, and a number of other useful procedures.

2.1.2 The Statistic Package

The statistics package provides functionality to compute a variety of statistical functions (e.g. summary statistics, histograms, etc.) on data. The `AbstractStatistic` class has been designed to tabulate data that is presented to its `collect()` method. The `AbstractStatistic` class provides a series of accessor methods that allow the user to retrieve information from the underlying statistics. These methods implement the `StatisticAccessorIfc` interface, which provides a read only view of the statistical information. Thus, users can sub-class from `AbstractStatistic` to more readily implement this interface. Figure 2 shows the basic functionality of the `AbstractStatistic` class. Instances can return the number of observations, the sample average, sample variance, the minimum, the maximum, etc. for the observations presented via the `collect()` method. The `collect()` method has been overridden to collect Boolean values (true = 1.0, false = 0.0) and can also collect weighted statistics.

Insert Figure 2 about here

The statistic package, as shown in Figure 3. allows for standard statistical collection through the `Statistic` class, as well as statistical collection algorithms designed for simulation data (e.g. standardize time series and batch means methods). Users can easily implement the `AbstractStatistic` base class to provide additional statistical collection algorithms.

Insert Figure 3 about here

2.1.3 Example

This section presents a simple example that utilizes the functionality of the random and statistic packages. For a single server queueing system, Lindley's equation, see Gross and Harris (1998), allows the computation of the waiting times of each of the customers based on knowledge of the arrival and service times. Let $X_1, X_2, \ldots$ represent the successive service times and $Y_1, Y_2, \ldots$ represent the successive inter-arrival times for each of the customers that visit the queue. Then, Lindley's equation is $W_{i+1} = \max(0, W_i + X_i - Y_i)$ where W_i is the waiting time of the ith customer. Code Listing 1 presents an implementation of a simulation based on Lindley's equation using constructs from the random and statistic packages.

Insert Code Listing 1 about here

In lines 3 and 4, the inter-arrival time and the service time distributions are defined. For illustrative purposes, the more general interfaces `RandomIfc` and `DistributionIfc` have been used. By default each distribution is given its own stream of random numbers when constructed. Thus, as

Accepted Pending Final Revisions

indicated in lines 3 and 4, there is no need to create a random number generator. Lines 6-8 specify the number of replications, number of customers, and the customer number to serve as a warm up period indicator. Lines 9-12 define the across replication statistics and the within replication statistics. For illustrative purposes, `avgw` has been declared as type `AbstractStatistic`. Thus, any sub-class of `AbstractStatistic` could have been used when creating the instance. The object `wbar` collects each customer's waiting time, while `avgw` collects the average waiting time for customers across replications. Line 16 implements Lindley's equation. Lines 17 and 18 collect the within replication statistics. Notice how, in line 18, the Boolean expression (`w > 0.0`) is observed. Because the collect method of the `Statistic` class treats true = 1.0 and false = 0.0, the statistics on this indicator are easily tallied. Lines 19-22 implement the resetting of the within replication statistics when the warm up indicator is reached. Lines 25 and 26 collect the end of replication averages for the waiting time and the probability of wait, before they are cleared/reset in lines 28-29. After all replications have been executed, the statistics are printed in lines 31-32. Exhibit 1 presents the sample statistical output from the program.

Insert Exhibit 1 about here

This program can be easily modified to use different distributions for the inter-arrival and service times by modifying lines 3 and 4. For example, by replacing line 3 with:

```
RngIfc rng = new AR1CorrelatedRng(0.8);
RandomIfc y = new Lognormal(1.0, 0.5, rng);
```

the inter-arrival times will be correlated lognormal random variables. Notice that the distribution takes in an instance of classes that implement the `RngIfc` interface. Thus, any correlation induction strategy can be easily accommodated simply by ensuring that it implements the `RngIfc`. Also note that the underlying random number streams (created within the distributions) continue for each replication from where they left off. Thus, each replication is independent. The streams for the distributions can be easily manipulated by using the `RandomStreamIfc` interface.

Code Listing 2 indicates how easy it is to perform an analysis using batch means and standardized time series. In this example, the within replication observations of the customer wait times from Lindley's equation are captured. Lines 1 and 2 declare instances of the `BatchStatistic` and `StandardizedTimeSeriesStatistic` classes to be used to directly observe the waiting time for each customer. Complete statistics based on these methods can easily be reported (not shown here due to space limitations). The default batching algorithm is essentially the same as th algorithm used in Arena™. See the discussion in Fishman (2001) or Kelton et al. (2006) for a description of the algorithm. The important point is that any simulation output analysis technique can be easily incorporated and used by adhering to the structure imposed by the framework's classes and interfaces.

Insert Code Listing 2 about here

Both the random and statistic packages have been designed to enable their use outside of the JSL as a general-purpose utility (e.g. for pure Monte-Carlo studies). The JSL also facilitates discrete-event modelling. The next section briefly describes the JSL's support for scheduling and executing events.

2.2 The Calendar Package

Within discrete-event simulation, the ability to schedule and execute events is a necessity. This ability is found within the `Scheduler` class and within the calendar package within the JSL. The `Scheduler` class only provides for processing the events (e.g. scheduling, executing the next event, cancelling events, updating simulated time, tracing events, etc.). The `Scheduler` class does not provide the data structure to hold events. The calendar package provides the data structures to hold events.

An event is a one-way transmission of information from one object to another that causes an *action* resulting in a change in state at a given instant in time. Within the JSL, this concept is represented within the `JSLEvent` class. `JSLEvent` holds a reference to an object that implements the `ActionListenerIfc` interface, which requires a method (`action(JSLEvent e)`) that supplies the actions associated with the event (in essence the event routine). Besides setting the time of the event, the user can set a priority, a name, and a type. In addition, the `JSLEvent` class has a reference to an `Object`, which can be used to send information along with the event when it is scheduled. `JSLEvent` implements the Java `Comparable` interface to provide an ordering first by smallest time, then by smallest priority, and finally by the order of creation.

Events are held within a data structure that is represented by the `CalendarIfc` interface. When constructing an instance of the `Scheduler` class an instance of a class that implements the `CalendarIfc` interface must be supplied. The calendar package provides four standard event calendar data structures based on a priority queue, a tree set, a skew heap, and a linked list. The default calendar is based on Java's `PriorityQueue` class. Users can easily supply their own calendar by implementing the `CalendarIfc` (`add(JSLEvent e)`, `nextEvent()`, `isEmpty()`, `cancelAll()`, `cancel()`, and `size()`) and using the `compareTo()` method of `JSLEvent` to properly order the data structure. Thus, the testing and use of different future event data structures can be easily accomplished within the JSL.

Accepted Pending Final Revisions

2.2.1 Scheduling and Executing Event Actions

As mentioned, the actions associated with an event should be contained within a class that implements the `ActionListenerIfc` interface. Within the JSL, this is usually accomplished by using an inner class that implements the `ActionListenerIfc` interface. Then, an instance of that class is supplied when scheduling the event. Code Listing 3 illustrates the event routines (in the form of action listeners) for a discrete event implementation of a simple M/M/1 queue.

Insert Code Listing 3 about here

Line 3 shows how events can be scheduled. An inherited `scheduleEvent()` method is used to schedule the arrival of the first customer. The object reference variable, `myArrivalListener`, represents an instance of the inner class, `ArrivalListener`, shown in lines 6-18. The object reference, `myArrivalRV`, is used to get the value of the time between arrivals when scheduling the event.

As shown in lines 6-18, the logic for handling an arrival to the system is placed in the action listener. This logic is very standard: if the number of busy servers is less than the number of available servers, then the number of busy servers is incremented and an end of service event is scheduled, else the number in queue is incremented (see lines 8-14.) Line 16 simply schedules the next arrival. Similar logic is presented in lines 20-30 for handling the end of service for a customer and scheduling the next customer to be served if there are customers waiting in the queue.

Code Listing 3 uses a number of object references, e.g. `myNumBusy` on lines 8 and 9. This object is a reference to a time weighted statistical collection variable represented by the `TimeWeighted` class. It is an example of a sub-class of `ModelElement`. The modelling of these types of objects within the JSL is accomplished with the modeling package and its sub-packages, which is discussed in the next section.

2.3 The Modeling and Model Element Packages

The key packages within the JSL for simulation modelling are the model package and its sub-package elements. The model package has classes that implement the functionality associated with simulation events and models. The elements package has classes that define objects that can be added to a simulation model (e.g. queues, resources, variables, etc.). The development of a simulation model is based on sub-classing the `ModelElement` class that provides the standard recurring actions within a simulation as well as allowing access to an instance of the `Scheduler` class.

Standard recurring actions are provided so that every model element can execute code prior to an experiment, prior to a replication, upon model initialization, upon warm up, after a replication, and after an experiment. For example, every sub-class of model element should implement an inherited protected method called `initialize()` to provide logic when the overall model is initialized that is specific to that model element. In Code Listing 3, lines 1-4 illustrate the `initialize()` method for the surrounding model element. This code is automatically called when the containing model is initialized at the start of each replication. For this example, this ensures that the first customer's arrival is scheduled when the model is initialized.

Insert Figure 4 about here

Figure 4 shows that a `ModelElement` may contain other instances of sub-classes of `ModelElement`. In addition, it shows that `Model` is a sub-class of `ModelElement` and that every `ModelElement` has a reference to a `Model`. The key to understanding and effectively using the JSL is to understand the recursive composite pattern implemented in Figure 4. It is helpful to conceptualize this type of association as a parent/child relationship. The parent is the composite (the thing that contains) and the children are the things that are contained. The composite pattern implies an object hierarchy. That is, a tree of objects. Within the JSL, an instance of `Model` serves as the base (root) of the tree, with each node being an instance of a sub-class of `ModelElement`. This allows model elements to be "hooked" into the model and then to react to standard simulation actions, such as initialization. This is a common pattern in simulation languages. For example, users of the simulation language Arena™ can implement visual basic for application code that occurs prior to and after a replication.

2.1.2 Building and Running a JSL Simulation Model

To build and run a JSL simulation model, the user must first design sub-classes of `ModelElement` that represent the modelling situation. Then, the user must create instances of their model elements and add them to an instance of a `Model`. For example, in Code Listing 4, line 4 creates an instance of `Model`, line 6 creates and instance of `JobShop` (a sub-class of `ModelElement`) and supplies the model as its parent. The `JobShop` model will be discussed further in the next section. Lines 8-12 perform this same procedure for the model element called `WorkStation`. In this manner, the model's object hierarchy is built. For readers that are familiar with other simulation languages, this is in concept similar to opening a model building window (creating a model) and dragging and dropping simulation constructs into the model. Within the JSL this process is code oriented; however, it should be clear that a graphical user interface could be constructed to perform such model building activities. Once a model has been constructed, it must be executed.

Code Listing 4 about here

The modeling package has three classes that facilitate the running of simulation models: `Replication`, `Experiment`, and `ExperimentRunner`. These classes are built upon an abstract base class that embodies the

Accepted Pending Final Revisions

notion of an iterative process (class IterativeProcess) in a general design pattern. One can think of an iterative process as an object-oriented implementation of the general notion of a do-while loop. An iterative process embodies a sequence of steps that can be executed. Prior to executing the steps, the iterative process must be initialized, then the steps can be executed, and finally some clean up code can be executed after the iterative process has been ended. Each step can be run individually or all steps can be run until a condition is met.

The class Replication is an iterative process that executes events until there are no more events or until some other stopping condition is met. Within the JSL, it is even possible to run experiments for a length of real clock time. The Experiment class is also an iterative process, but its steps are instances of the class Replication. Thus, an experiment is simply a sequence of replications. The user has full control over setting the length of the replications, the length of the warm up period, running until a condition is set (e.g. a half-width is reached), etc.

In Code Listing 4, line 34 constructs an instance of an Experiment, passing in a reference to the model. Then, in lines 36-38, the experiment's running conditions are set. Line 40 turns on some default text file reporting to capture all statistical information from the experiment. Finally, line 42 tells the experiment to run all of its replications via the method runAll().

Individual (single) instances of the class Replication can also be run in this manner. Finally, the class ExperimentRunner is also an iterative process whose steps are instances of the class Experiment. The user provides an iterator that returns experiments, which then will each be run in turn until all the experiments have been executed. In this manner, a set of experimental scenarios can be easily constructed and run within the JSL.

The JSL has a large number of already constructed model elements for use in simulations. Due to space limitations, only a few are described in this paper.

- Variable – A class that represents the use of variables within a model. Variables have an initial value and will be reset to the initial value automatically at the beginning of each replication.
- RandomVariable – A sub-class of Variable that can take on a random value within the simulation model. The JSL handles stream control for its random variables. At the end of each replication, the streams of each random variable are advanced to the next sub-stream; thereby, improving synchronization for the purpose of using common random numbers.
- ResponseVariable – A sub-class of Variable that provides observation-based statistical collection.
- TimeWeighted – A sub-class of ResponseVariable that facilitates the use of time-based variables. As indicated in Code Listing 3, these variables can be incremented (line 13) and decremented (line 24). This enables the automatic collection of time-weighted statistics on the variable.
- Counter – A sub-class of ResponseVariable that facilitates the definition of a variable that can be used to count occurrences and to stop the simulation when the counter reaches a limit. Other actions can also be triggered when the counter reaches its limit.
- ResponseVariableAverageObserver, AggregateTimeWeightedVariable, AggregateCounter – These classes are examples that allow for the automatic collection of aggregate statistics across responses within a simulation. For example, by attaching an instance of AggregateTimeWeightedVariable to multiple instances of TimeWeighted variables an aggregate time weighted total is collected. This is facilitated through the use of the Observer/Observable pattern. Whenever, the value of the variable changes (e.g. line 24), any listeners will be notified.
- Queue – This class models a queue for holding objects that must wait within a simulation and provides for automatic collection of time in queue and number in queue statistics. The Queue model element automatically empties any queued entities when it is initialized at the beginning of a replication. The runtime switching of queue disciplines is provided within the framework.
- Resource – This class models a resource having a common set of units that can be allocated and automatically tabulates resource usage statistics.
- ResourceSet – This class models a set of resources that may be requested via various resource selection rules.
- EventGenerator – This class allows the user to define a recurring pattern of events, specify the time between events, the number of events to occur, etc. This provides the functionality similar to a CREATE module found in many simulation languages.

In addition to these model elements, the JSL has a package that facilitates the modelling of objects that move through space. For example, the user defines a spatial model (essentially a coordinate reference frame) and uses model elements that implement the SpatialElementIfc interface. The modelling of mobile resources is facilitated with classes (MobileResource, Transporter, TransporterSet, etc.). In addition, as mentioned in the literature review section, the JSL has packages for modelling queuing networks, inventory systems, supply chains, and transportation networks.

Once a JSL model has been built the user can register observers to any model element within the model. These observers can be notified when the model element indicates a change. This mechanism is facilitated within the JSL via the observers package.

Accepted Pending Final Revisions

2.4 The Observers Package

The observers package within the JSL is predicated on exploiting, to the extent possible, the already existing Observer/Observable pattern (see Gamma et al. (1995)) implemented within the Java system libraries. With the Observer/Observable pattern, the data is separated from the view of the data. In this pattern, an observable object (an instance of a class that extends the Java class `Observable`) can be observed by one or more objects called Observers, which implement the `Observer` interface. The `Observable` class has methods that allow the registration and management of observers. `ModelElement` extends from `Observable` and thus inherits all of its methods. Sub-classes of `ModelElement` can use the `notifyObservers()` method to notify any registered observers that it has changed in some manner. This pattern allows for loose coupling between the `Observer` and the `Observable` classes. In fact, the `Observable` object does not need to know the type of the object that is observing it. An observer could be a graphical user interface component, a class that writes to a database, a statistical collection object, etc. This provides for extreme flexibility when extending and developing models based on the JSL.

Statistical collection and output reporting are implemented using the observers package within the JSL. The previously mentioned classes (`ResponseVariable` and `TimeWeighted`) both are observed by instances of the `StatisticalObserver` class. In this case, whenever the value of the variable changes, the instance of a `StatisticalObserver` is notified. It then is responsible for tabulating the appropriate statistics (batch, within replication, across replication, etc.)

The observer package has many other classes that observe parts of the JSL. Some examples include:

- `TimeWeightedObserver` – This class observes and collects statistics on TimeWeighted variables.
- `VariableTraceTextReport` – This class can be used to write the value of variables and the time that the variable changes to a text file.
- `SchedulerTraceReport` – This class is used by the `Scheduler` class to write each event to a text file for tracing events.
- `BatchReport` – This class facilitates the reporting in a text file the replication batch statistics.
- `ExperimentReport` – This class reports in a text file statistics and information about the running of an experiment.
- `ReplicationReport` – This class is used to report information to a text file about the running of a replication and its statistics.
- `AccessDBReport`, `AccessDBBatchReport`, `AccessDBReplicationReport`, `AccessDBSummaryReport` – These classes provide examples for how to write the statistical information collected during a simulation (e.g. batch statistics, replication statistics, experiment statistics, etc.) to a Microsoft Access™ database. Since Java facilitates the usage of any database through its Java database connectivity functionality, any database can be easily used.

Of course, because there can be multiple observers on the same element, additional observers (e.g. user interface dials, animation, etc) can be easily registered to meet a variety of user needs. Thus, the framework provided by the JSL can be readily extended.

The next section presents an example of a relatively interesting model to illustrate how to model with the JSL.

3 JOB SHOP SIMULATION EXAMPLE

This example is based on the job shop model in Law (2007) page 140. A job shop is essentially a queueing network where the jobs are the customers that must visit a sequence of workstations in order to get some processing completed. Each workstation has a queue and a resource with a specified capacity. A sequence consists of a number of job steps, with each job step representing the machine and the processing time distribution for the given step. There can be a number of different job types in the system, with each job type being assigned a particular sequence to follow through the shop. Jobs that arrive to the shop are assigned a particular job type according to a probability distribution and then the jobs begin to follow the sequence associated with that job type. The system time (total time from creation to exiting the system) must be collected by job type.

For this example, there are 5 workstations in the shop and the job arrival process is Possion with a mean rate 4 per hour. The distribution of job types is (0.3, 0.5, and 0.2) for the respective job types 1, 2, and 3. There are 3 sequences, with job type 1 requiring workstations (3, 1, 2, 5), job type 2 requiring workstations (4, 1, 3), and job type 3 requiring workstations (2, 5, 1, 4, 3). Jobs that arrive to a workstation require 1 unit of the resource and if it is not available will wait in a first in first out queue. The service times at each station are all Erlang order 2 random variables with the mean depending on the type of job and the current station.

To model this situation using the JSL it is useful to conceptualize the classes of objects within the system and how they relate. Then, the model can be built by developing new sub-classes of `ModelElement` and/or by directly using already existing classes. Figure 5 presents a class diagram for the job shop model. The class `JobShop` represents the entire system and can hold instances of `WorkStation`, `Sequence`, and `JobGenerator`. The `WorkStation` class represents the location within the job shop where processing occurs and as indicated in the figure consists of an instance of `Queue` and an instance of a `Resource`. The class `Sequence` represents the sequence of workstations to be visited using job steps (represented by the class `JobStep`). The class `JobStep` knows the workstation and a reference to an object that implements the

Accepted Pending Final Revisions

RandomIfc. This reference will be used to hold the service time at the workstation. The class, JobGenerator, is responsible for creating the jobs and introducing them into the model. The class JobGenerator uses an instance of RandomList (part of the random package) to allow for randomly assigning the type of job. The class JobType represents the notion that jobs have specific types and will collect system time responses by job type via the relationship to ResponseVariable.

Insert Figure 5 about here

The following presents part of the implementation of this model; however, many details have been omitted due to limitation on space. From a simulation modelling standpoint, the JobGenerator, WorkStation, JobType, and Job classes are the most interesting. Thus, the discussion will concentrate on these classes.

The Job class is like the concept of an entity found in many simulation languages. It represents the things (jobs) that flow through the system. Code Listing 5 presents and excerpt from the JobGenerator class. Both the Job and JobType classes are implemented as inner classes within JobGenerator. Lines 10-14 show that a job type has a name, a sequence, and a response variable associated with it. The class Job, shown in lines 16-38, shows that they are a sub-class of Request and that each job knows its job type (line 17) and has an iterator to represent its process plan (line 18). A Request is used when interacting with the Resource class to represent the thing that uses the resource. This iterator is determined by the job's sequence via line 23. The job type is determined randomly in line 22. The object reference, myJobTypes, is an instance of a RandomList that returns job types according to a user specified probability distribution across the elements in the list.

Code Listing 5 about here

JobGenerator is a sub-class of EventGenerator (not shown). Every instance of an EventGenerator must have a generate() method, see lines 1-8. The generate() method is called every time the event occurs for EventGenerator. As can be seen in line 4, this method creates an instance of a Job and then tells the job to perform its next job step (line 6). As per lines 27-37, when a job performs its next job step it first checks to see if it has another job step (via line 29). Then, in lines 30-33, it gets the appropriate step, sets the service time for the step (line 31) and then tells the workstation associated with the step to process the job (line 33). If there are no more job steps, then line 35 is invoked to collect statistics on the total time spent by the job in the system.

Code Listing 6 about here

The WorkStation class performs in a similar manner as previously described for the single server queue code described in Code Listing 3. Code Listing 6 presents the WorkStation class in its entirety. In line 1, WorkStation extends from the class SchedulingElement. This class provides convenient methods within the JSL for scheduling events. The structure of the class is very standard. First, in lines 3-5 the necessary object references are declared. Then in lines 13-15, the objects are created. For example, line 14 creates an instance of a Resource with the workstation serving as the resource's parent model element. Lines 18-28, implement the logic to handle the arrival of jobs to the workstation. This is similar to the previously mentioned customer arrival logic. In this case, the job is first enqueued in line 19. Then, if the resource is idle, the job is removed, the units of the resource allocated to the job (line 24), and then the job is started into service. Lines 36-47 implement the end of service logic. In line 38, the currently departing job is retrieved as an object attached to the event. Then, it is used in line 39 to release the resource. If there is a waiting job, then lines 40-44 remove the job and start the job into service. Finally, the departing job is asked to perform its next job step in line 45. This sends the job to the next station in its sequence.

Code Listing 4 already illustrated how to construct and run this model. Recall that each of the individual elements of the model (e.g. JobGenerator, WorkStation, etc.) must be created and associated with an instance of the Model class. Then, an instance of the experiment can be made and executed. Running the model produces statistical output for each queue (time in queue and number in queue), each resource's number busy statistics, system time by job type, etc.

Table 1 presents the results (sample average (Avg.) and 95% confidence interval half-widths (HW)) from running the model for 100 replications with a warm up period of 5000 hours and a run length of 10000 hours. These settings were chosen arbitrarily for illustrative purposes. The same situation was modelled using Arena™ and executed under the same settings. As can be seen from the table, the results are essentially the same. No formal statistical test was done because it is well known that if enough samples are collected a statistical test of the difference can be made to reject the hypothesis of equality because each system was run under different random numbers. While no formal testing of time to execute was examined, it is interesting to note that the Arena™ model took 2.85 minutes to execute while the Java program took 3.26 minutes on the same computer.

2.1.3 Process View Modelling

To wrap up the modelling capabilities of the JSL, an overview of how the JSL handles process view modelling is presented. In Rossetti et al. (2000) a prototype for using the Java language's thread mechanism to implement the process interaction approach within the JSL was discussed. The initial testing of this concept proved disappointing. Because of the way that Java implements its threading, the constant

Accepted Pending Final Revisions

context switching that occurs within a simulation model makes the overhead of using threads in this manner very problematic. In fact, L'Ecuyer et al. (2002) describe this similar problem in their thread-based implementation for processes. Simulations using Java's thread mechanism take significantly longer to execute. Even Java's recent revisions of its thread packages have not made an improvement in this regard. The fact is, Java's thread mechanism was never meant for this use. Weatherly and Page (2004) also discuss this issue and present a very reasonable argument for adding co-routine support to the Java language. Co-routines are the basic mechanism that the languages Simscript™ and ModSim use for their process interaction approach to simulation modelling. Indeed, this would be great improvement for the use of Java for simulation modelling if it was realized. Jacobs and Verbraeck (2004) also discuss the issues related to using threads as the underlying mechanism for performing process interaction in Java. They develop formalisms for mapping process routines on to the event scheduling formalism via a Java interpreter. Their approach is based on class reflection.

For the moment, the original implementation of the JSL's process interaction approach based on Java's threads has been depreciated; however, the JSL does support the process description approach to implementing the process view. The approach taken by the JSL is similar (in spirit) to what was done in Jacobs and Verbraeck (2004). The JSL maps the process interaction to a state transition diagram that functions based on event scheduling. The advantage of the JSL's mechanism is that it does not require an interpreter. The disadvantage is that entity state and its relationship to the process state is more difficult to maintain. For example, the JSL does not store context like a co-routine implementation does.

Simulation languages like Arena™ are built upon the notion of describing the process (or life of an entity) through a series of commands. This is often built via a network representation of the flow of entities as described in Joines and Robert (1996) (e.g. YANSL). The JSL implements the process view by allowing the user to describe the flow of an entity through a series of commands. Essentially, the commands are held in a list and the current command that the entity is executing is always noted so that the next command can be executed. This is similar to Jacobs and Verbraeck's notion of a control state. The user uses the `ProcessDescription` class to add various commands (`ProcessCommands`) to the process. Then, the client can create a process executor (`ProcessExecutor`) that will iterate through the commands for a given entity. This executes the process for the entity.

The `ProcessExecutor` is the real workhorse in implementing the JSL's process description approach. In essence, an instance of a `ProcessExecutor` acts as a separate "process" that runs an entity through the list of commands associated with a particular `ProcessDescription`. The `ProcessExecutor` keeps track of the current state of the "process", i.e. whether it is created, initialized, suspended, executing, or terminated. In addition, a `ProcessExecutor` keeps track of the currently executing command, which enables the suspension, resumption, jumping within, and the termination of the execution of the commands. The `ProcessExecutor` relies heavily on a rigorous state pattern, Gamma et al. (1995) (see Figure 5) to properly implement the legal processing of the commands. The key to this pattern is the ability to suspend and resume the iteration through the process commands. In the figure, when an instance of a `ProcessExecutor` is constructed it is immediately placed in the Created state. Then, it may only be initialized. Once in the Initialized state, it is ready to be executed. This is accomplished with the use of the `start(command index)` method, which tells the executor to start executing the command at the provided index. Once in the Executing state, the process can transition normally to the next command in the process description or jump to other commends. In addition, the process, can be suspended at the current command. If the process has been suspended, it can then be resumed at a particular command. Finally, the process can be terminated. Once terminated the process can be re-used again by using the `initialize()` method.

Insert Figure 5 about here

Due to space limitations, the coding details of how all this is implemented are omitted; however, Code Listing 7 shows how easy it is to develop a model using these constructs.

Insert Code Listing 7 about here

In lines 4-17 of the listing, the major model elements necessary to model job type 2's process within the job shop model are constructed and added to the model. In lines 12-17, an instance of a `Resource` and an instance of a `Queue` are used to represent a workstation in the job shop. Then, in line 19, an instance of a `ProcessDescription` is created. This will be used to describe the life of a type 2 job. In line 23, an instance of an `EntityProcessGenerator` is created. This class will create the entities according to an inter-arrival time and instantiate an instance of the `ProcessExecutor` class to have the entity execute the process described by the `ProcessDescription` class.

In lines 26-36, a series of process commands (e.g. `Seize`, `Delay`, and `Release`) are created and attached to the instance of `ProcessDescription`. These represent the sequence of visits to workstations 4, 1, and 3 required by job type two. When the simulation model is run, entities will be created and flow through these commands according to the process description.

Since the JSL is open source, anyone can add to the features of the process modelling by implementing additional process commands. The JSL process description architecture also supports the use of macro commands (commands consisting of other commands). For example, the three commands `Seize`, `Delay`, and `Release` can be combined into a single command (e.g. like Arena™'s Process module) without much difficulty.

Accepted Pending Final Revisions

4 FUTURE RESEACH AND DEVELOPMENT

The JSL represents a readily extendable and useful open source software implementation for developing discrete event simulation models. The only code that is not licensed under the GNU GPL is the random number generator from L'Ecuyer (2001). This code can be found at (http://www.iro.umontreal.ca/~simardr/indexe.html) and is freely available for non-commercial purposes.

Because of the JSL's extendibility, there always remains room for future software development. The key areas for future development include 1) adding to the range of statistical output techniques, 2) adding to the variety of random number distributions, 3) providing distribution fitting utilities, 4) adding more process commands to the process description modelling elements, 5) re-implementing the process view based on threads for educational and testing purposes, 6) adding animation, and 7) adding graphical user interface support.

Future research within the JSL involves adding to the modelling capabilities for supply chains, queueing networks, and transportation networks. Other addition research work involves adding support packages for simulation optimization techniques, studying the efficiency and memory usage of different JSL design implementations, and integrating other simulation paradigms into the JSL architecture (e.g. agent based modelling and continuous simulation).

A number of tutorial examples for the JSL have been developed based on examples within Law (2007). In particular, there are implementations of the time-shared computer model, the (s, S) inventory system, the job shop model, and other queuing models. Thus, the JSL would be an excellent complement to the use of Law (2007) for teaching simulation. A complete set of JavaDocs has been generated for the library, and of course, the source is available. The author also plans a textbook based on the JSL.

REFERENCES

Banks, J., Carson, J., Nelson, B., and Nicol, D. (2005) *Discrete-Event System Simulation,* 4th, Edition, Prentice-Hall.

Biller, B. and Nelson, B. L. (2003) "Modelling and Generating Multivarate Time-Series Input Processes Using a Vector Autogressive Technique, *Assoc. Comput. Mach. Trans. Modelling and Comput. Simul.*, 13, 211-237.

Booch, G., J. Rumbaugh, and I. Jacobson. (1999) *The Unified Modelling Language User Guide.* Addison-Wesley.

Borshchev, A. V., Kolesov, Y. B. and Senichenkov, Y. B. (2000) "Java Engine for UML Based Hybrid State Machines", in *Proceedings of the 2000 Winter Simulation Conference*, ed. J. Joines, R. Barton, P. Fishwick, and K. Kang, Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, pp. 1888-1894.

Cario, M. C., and Nelson, B. L. (1996) "Autoregressive to Anything: Time Series Input Processes for Simulation", *Operations Research Letters*, 19, 51-58.

Cario, M. C. and Nelson, B. L. (1998) "Numerical Methods for Fitting and Simulating Autoregressive-To-Anything Processes", *INFORMS Journal of Computing,* 10, 72-81.

Codl, D., Kacer, J. and Koutny, T. (2003) "Comparison-Evaluation of Java-Based Discrete-Time Simulation Tools, in *Proceedings of the 37th International Conference* MOSIS-2003: Modeling and Simulation of Systems, ed. Stefan, J., pp. 125-130, MARQ, Ostrava, Czech Republic, April.

Fishman, G. S. (2001) *Discrete-Event Simulation: Modeling, Programming, and Analysis*, Springer, New York.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.

Garrido, J. M. (2001) Object-Oriented Discrete-Event Simulation in Java, Kluwer Academic/Plenum Publishers, New York.

Gross, D. and Harris, C. M. (1998) *Fundamentals of Queueing Theory*, 3rd Edition, John Wiley & Sons, New York.

Healy, K. J. and R. A. Kilgore. (1997) "Silk: A Java-Based Process Simulation Language" *Proceedings of the 1997 Winter Simulation Conference*, ed., S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 475-482, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey

Hobbs, B., Rossetti, M. D., Faas, P. (2006) "An object-oriented framework for simulating automatic data collection systems", in *Proceedings of the 2006 Winter Simulation Conference*, L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds., Piscataway, New Jersey: Institute of Electrical and Electronic Engineers .

Jacobs, P. H. M., N.A. Lang, A. Verbraeck. (2002) "D-SOL: A distributed Java based discrete event simulation architecture". In *Proceedings of the 2002 Winter Simulation Conference*, E. Yücesan, C.-H. Chen, J.L. Snowdon and J.M. Charnes (Eds.). San Diego, 8-11 December 2002. IEEE, San Diego, CA, pp. 793-800.

Jacobs, P. H. M. and Verbraeck, A. (2004) "Single-Threaded Specification of Process-Interaction Formalism in Java", In *Proceedings of the 2004 Winter Simulation Conference*, R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds., Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Joines, J.A. and S. D. Roberts. (1996) Design of object-oriented simulations in C++. In *Proceedings of the 1996 Winter Simulation Conference*, ed., J. Charnes, D. Morrice, D. Brunner, and J. Swain, 65-72. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Kacer, J. (2001) "J-Sim – A Java-based Tool for Discrete Simulations, in *Proceedings of the 23rd International Autumn Colloquium*, ASIS-2001, pp. 135-141, Advanced Simulation of Systems MARQ, Ostrava, Czech Republic, September.

Kelton, W. D., Sadowski, R. P., and Sturrock, D. T. (2004) *Simulation with Arena*, 3rd Edition, McGraw-Hill.

Law, A. (2007) *Simulation Modelling and Analysis*, 4th Edition, McGraw-Hill

L'Ecuyer, P. (2001) "Software for uniform random number generation: Distinguishing the good and the bad", in *Proceedings of the 2001 Winter Simulation Conference*, ed. B. A Peters, J. S. Smith, J. Medeiros, and M. W. Rohrer, Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, pp. 95-105.

L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002) "SSJ: A framework for stochastic simulation in Java", in *Proceedings of the 2002 Winter Simulation Conference*, E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds., Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.

Marzolla, M., (2004) "libcppsim: a Simula-like, Portable Process-Oriented simulation library in C++", *Proceedings of ESM'04, the 18th European Simulation Multiconference* (Graham Horton, editor), Magdeburg, DE, Jun 13-16 2004, ISBN 3-936150-35-4, pp. 222-227

Page, B. and Kreutzer, W. (2005) *The Java Simulation Handbook: Simulating Discrete Event Systems with UML 2 and Java.* Shaker, Aachen.

Accepted Pending Final Revisions

Rossetti, M. D., Aylor, B., Jacoby, R., Prorock, A., White, A. (2000) "Simfone′: An object-oriented simulation framework", in *Proceedings of the 2000 Winter Simulation Conference*, ed. J. Joines, R. Barton, P. Fishwick, and K. Kang, Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, pp. 1855-1864

Rossetti, M. D., and Chan, H. T. (2003) "A Prototype Object-Oriented Supply Chain Simulation Framework", in *Proceedings of the 2003 Winter Simulation Conference*, S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds., Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.

Rossetti, M. D. and Nangia, S. (2007) "An object-oriented framework for simulating full truckload transportation networks", in *Proceedings of the 2007 Winter Simulation Conference*, S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds., Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rossetti, M. D., Miman, M., Varghese, V., and Xiang, Y. (2006) "An object-oriented framework for simulating multi-echelon inventory systems", in *Proceedings of the 2006 Winter Simulation Conference*, L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds., Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rossetti, M. D. and Thomas, S. (2006) "Object-Oriented Multi-Indenture Multi-Echelon Spare Parts Supply Chain Simulation Model", *International Journal for Modelling and Simulation*, Vol. 26, No. 4.

Sánchez, P. J. (2006) "As simple as possible, but no simpler: a gentle introduction to simulation modelling", in *Proceedings of the 2006 Winter Simulation Conference,* L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds., Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.

Schwetman, H., (1986) "CSIM++: A C-Based, Process-Oriented Simulation Language," in *Proceedings of the 1986 Winter Simulation Conference*, ed., J. Wilson, J. Henrikson, S. Roberts, 386-396, Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Weatherly, R. and Page, R. M. (2004) "Efficient process interaction simulation in Java: Implementing co-routines with a single Java thread", In *Proceedings of the 2004 Winter Simulation Conference*, R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds., Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

WEBSITES

The open source web site will be given here

Accepted Pending Final Revisions

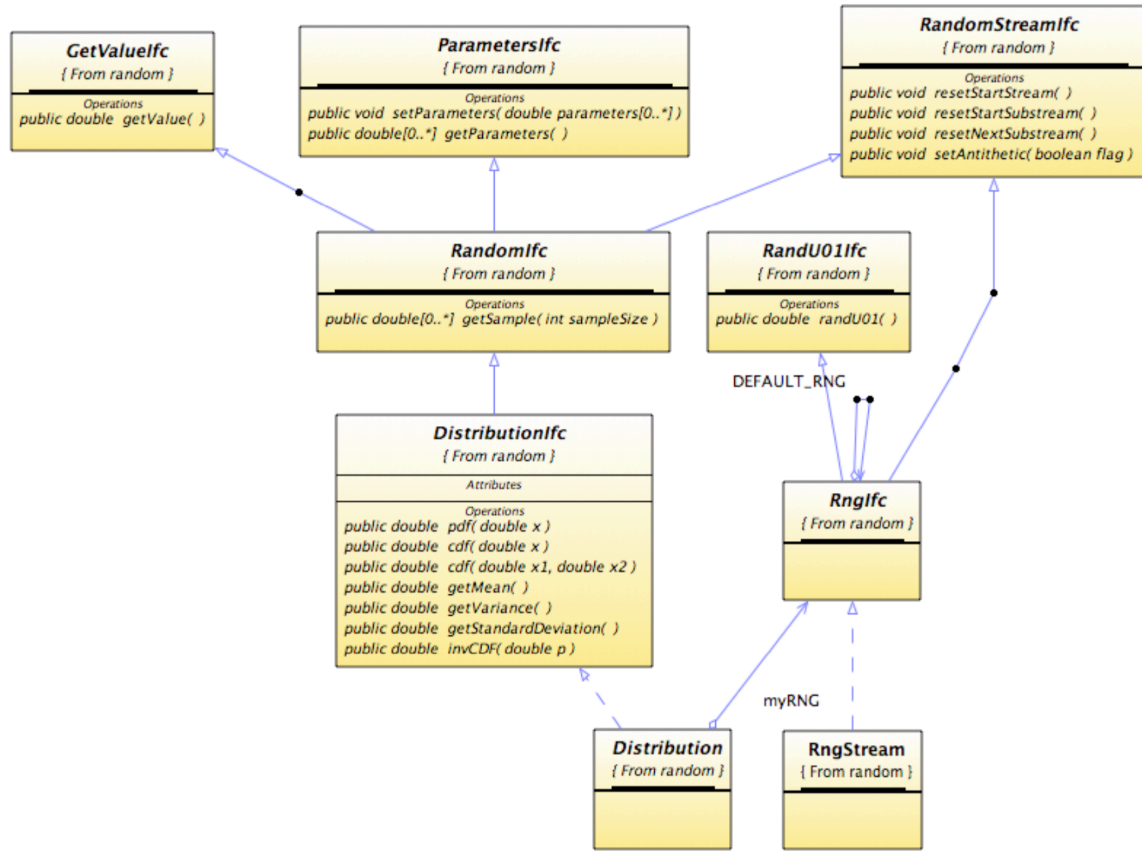

Figure 1: Key Interfaces and Classes in Random Package

Accepted Pending

Copyright © 200? Inderscience Enterprises Ltd.

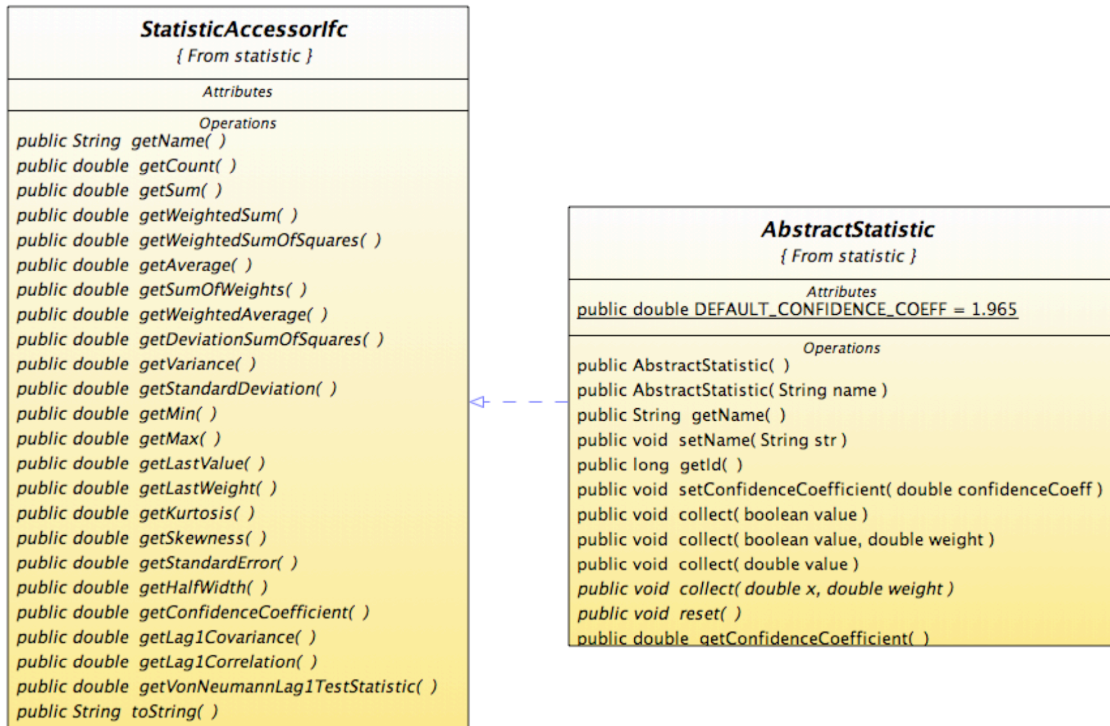

Figure 2: `AbstractStatistic` and its Read-Only Interface

Accepted Pending Final Revisions

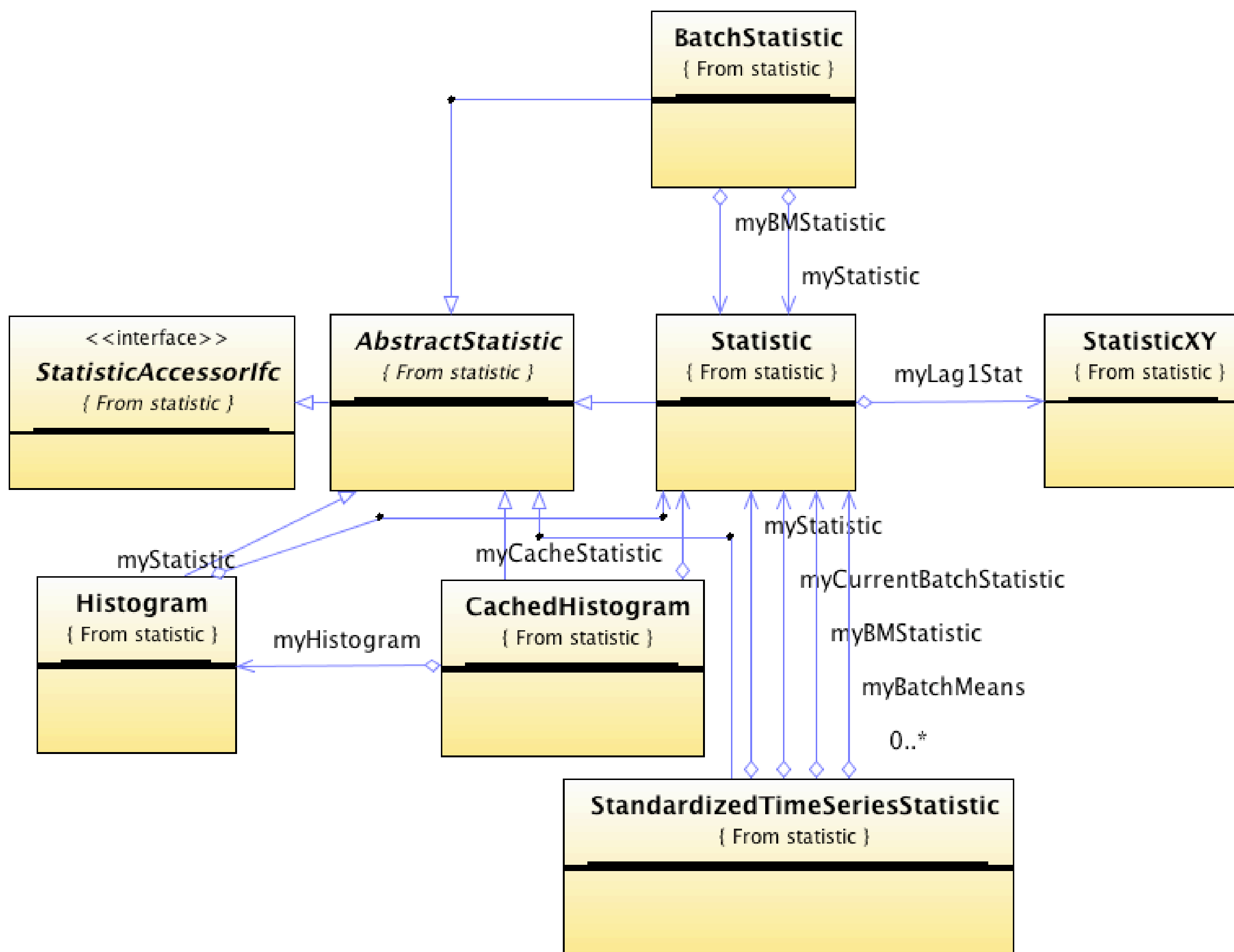

Figure 3: Classes in the Statistics Package

Accepted Pen

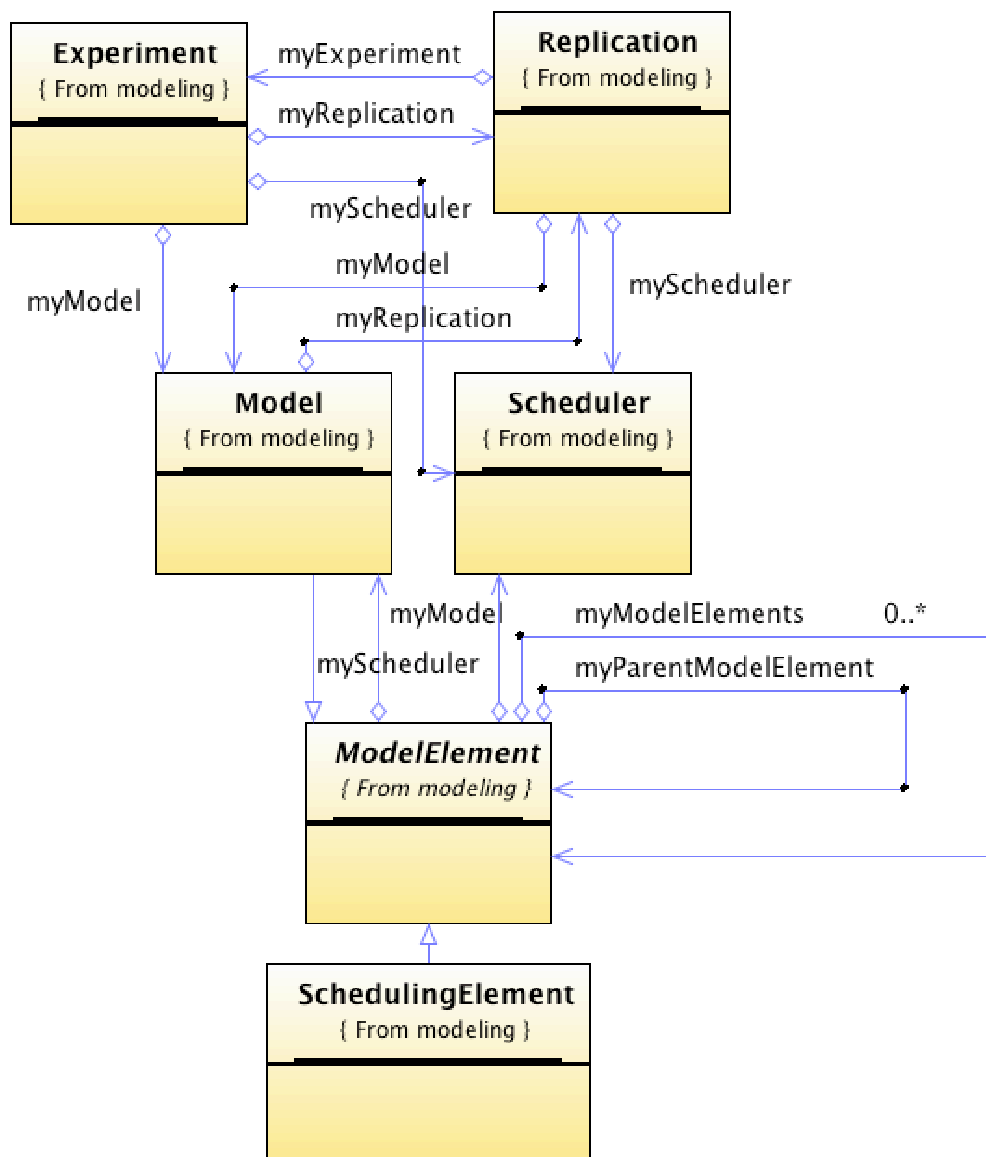

Figure 4: Key Modelling Classes

Accepted ... ons

Code Listing 1: Lindley's Equation Example

```
1  public static void main(String[] args) {
2    // inter-arrival time distribution
3    RandomIfc y = new Exponential(1.0);
4    // service time distribution
5    DistributionIfc x = new Exponential(0.7);
6    int r = 30; // number of replications
7    int n = 100000; // number of customers
8    int d = 10000; // warm up
9    AbstractStatistic avgw = new Statistic(“Avg waiting time");
10   Statistic avgpw = new Statistic("prob of wait");
11   Statistic wbar = new Statistic("Within rep avg waiting time");
12   Statistic pw = new Statistic("Within rep prob of wait");
13   for(int i=1;i<=r;i++){
14      double w = 0; // initial waiting time
15      for(int j=1;j<=n;j++){
16        w = Math.max(0.0, w + x.getValue() - y.getValue());
17        wbar.collect(w);// collect waiting time
18        pw.collect((w>0.0)); // collect P(W>0)
19        if (j == d){// clear stats at warmup
20              wbar.reset();
21              pw.reset();
22        }
23      }
24      //collect across replication statistics
25      avgw.collect(wbar.getAverage());
26      avgpw.collect(pw.getAverage());
27      // clear within replication statistics for next rep
28      wbar.reset();
29      pw.reset();
30   }
31   System.out.println(avgw);
32   System.out.println(avgpw);
33 }
```

Accepted Pending Final Revisions

Exhibit 2: Sample Output for Lindley Equation Example

```
ID 1
Name Across rep avg waiting time
Number 30.0
Minimum 1.5523842361703213
Maximum 1.7601221186157152
Sum 49.42075879668269
Average 1.6473586265560898
Variance 0.001860191056515051
Standard Deviation 0.04312993225725089
Standard Error 0.007874412266988673
Confidence Coefficient 1.965
Half-width 0.015473220104632743
Weighted Average 1.64735862655609
Weighted Sum 49.4207587966827
Sum of Weights 30.0
Weighted Sum of Squares 81.46765887530192
Last value collected 1.6325147072873045
Last weighted collected 1.0
Lag 1 Correlation -0.14736789218428864
Von Neumann Lag 1 Test Statistic -0.8121440898730583
Number of missing observations 0.0

ID 2
Name Across rep prob of wait
Number 30.0
Minimum 0.6895666666666654
Maximum 0.7112333333333374
Sum 21.026577777777817
Average 0.7008859259259272
Variance 1.563895217823943E-5
Standard Deviation 0.003954611507877788
Standard Error 7.220099763447277E-4
Confidence Coefficient 1.965
Half-width 0.001418749603517389
Weighted Average 0.7008859259259272
Weighted Sum 21.026577777777817
Sum of Weights 30.0
Weighted Sum of Squares 14.737685964444502
Last value collected 0.7009333333333216
Last weighted collected 1.0
Lag 1 Correlation -0.1389880977659123
Von Neumann Lag 1 Test Statistic -0.7830506130522926
Number of missing observations 0.0
```

Accepted Pending Final Revisions

Code Listing 2: Batching and Standardized Time Series Analysis

```
1   AbstractStatistic wbar = new BatchStatistic("Batch waiting
    time");
2   AbstractStatistic wbarSTS = new
    StandardizedTimeSeriesStatistic("STS waiting time");
3   double w = 0; // initial waiting time
4   for(int j=1;j<=n;j++){
5      w = Math.max(0.0, w + x.getValue() - y.getValue());
6      wbar.collect(w);// collect waiting time
7      wbarSTS.collect(w);
8      if (j == d){// clear stats at warmup
9         wbar.reset();
10        wbarSTS.reset();
11     }
12  }
```

Accepted Pending Final Revisions

Code Listing 3: Example Event Routines

```
1  protected void initialize() {// start the arrivals
2    super.initialize();
3    scheduleEvent(myArrivalListener, myArrivalRV.getValue());
4  }
5
6  class ArrivalListener implements ActionListenerIfc {
7    public void action(JSLEvent event) {// new customer arrived
8      if (myNumBusy.getValue() < myNumServers) { // server available
9        myNumBusy.increment(); // make server busy
10       //  schedule end of service
11       scheduleEvent(myEOSListener, myServiceRV.getValue());
12     } else { // no server available
13       myNQ.increment(); // place customer in queue
14     }
15     //  always schedule the next arrival
16     scheduleEvent(myArrivalListener, myArrivalRV.getValue());
17   }
18 }
19
20 class EndServiceListener implements ActionListenerIfc {
21   public void action(JSLEvent event) {// customer departing
22     myNumBusy.decrement(); // customer is leaving server is freed
23     if (myNQ.getValue() > 0) { // queue is not empty
24       myNQ.decrement(); // remove from queue
25       myNumBusy.increment(); // make server busy
26       //  schedule end of service
27       scheduleEvent(myEOSListener, myServiceRV.getValue());
28     }
29   }
30 }
```

Accepted Pending Final Revisions

Code Listing 4: Building and Running the Job Shop Model

```
1  public static void main(String[] args) {
2    System.out.println("Jobshop Example");
3    // create the containing model
4    Model m = Model.createModel();
5    // create the jobshop
6    JobShop shop = new JobShop(m, "JobShop");
7     // create the workstations
8    WorkStation w1 = shop.addWorkStation(3, "w1");
9    WorkStation w2 = shop.addWorkStation(2, "w2");
10   WorkStation w3 = shop.addWorkStation(4, "w3");
11   WorkStation w4 = shop.addWorkStation(3, "w4");
12   WorkStation w5 = shop.addWorkStation(1, "w5");
13    // create the sequences
14    Sequence s1 = shop.addSequence();
15   s1.addJobStep(w3, new Gamma(2.0, 0.5/2.0));
16   s1.addJobStep(w1, new Gamma(2.0, 0.6/2.0));
17   s1.addJobStep(w2, new Gamma(2.0, 0.85/2.0));
18   s1.addJobStep(w5, new Gamma(2.0, 0.5/2.0));
19   Sequence s2 = shop.addSequence();
20   s2.addJobStep(w4, new Gamma(2.0, 1.1/2.0));
21   s2.addJobStep(w1, new Gamma(2.0, 0.8/2.0));
22   s2.addJobStep(w3, new Gamma(2.0, 0.75/2.0));
23   Sequence s3 = shop.addSequence();
24   s3.addJobStep(w2, new Gamma(2.0, 1.2/2.0));
25   s3.addJobStep(w5, new Gamma(2.0, 0.25/2.0));
26   s3.addJobStep(w1, new Gamma(2.0, 0.7/2.0));
27   s3.addJobStep(w4, new Gamma(2.0, 0.9/2.0));
28   s3.addJobStep(w3, new Gamma(2.0, 1.0/2.0));
29   JobGenerator jg = shop.addJobGenerator(new Exponential(0.25));
30   jg.addJobType("A", s1, 0.3);
31   jg.addJobType("B", s2, 0.5);
32   jg.addLastJobType("C", s3);
33   // create the experiment to run the model
34   Experiment e = new Experiment(m, "Job Shop");
35   // set the parameters of the experiment
36   e.setNumberOfReplications(100);
37   e.setLengthOfReplication(10000.0);
38   e.setLengthOfWarmUp(5000.0);
39   // turn on the desired reporting
40   e.turnOnExperimentReport();
41   // tell the experiment to run
42   e.runAll();
43   System.out.println("Done!");
44 }
```

Accepted Pending Final Revisions

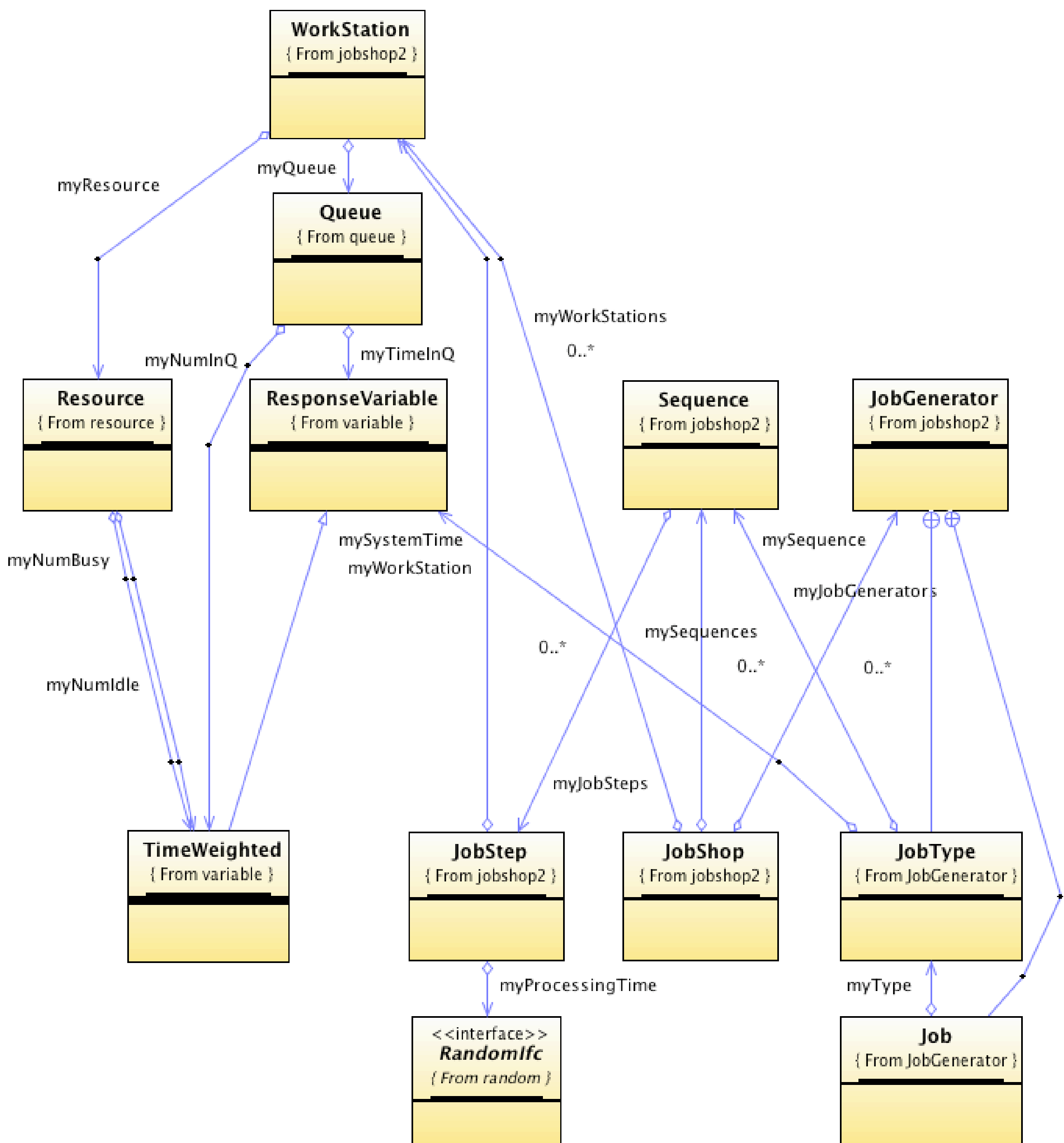

Figure 5: Class Diagram for Job Shop System

Code Listing 5: Generating Jobs in the Job Shop Model

```
1  protected void generate(JSLEvent event){
2    if (!myJobTypes.isEmpty()){
3      // create the job
4      Job job = new Job();
5      // tell it to start its sequence
6      job.doNextJobStep();
7    }
8  }
9
10 class JobType {
11   String myName;
12   Sequence mySequence;
13   ResponseVariable mySystemTime;
14 }
15
16 class Job extends Request {
17   JobType myType;
18   Iterator<JobStep> myProcessPlan;
19
20   Job(){
21     super();
22     myType = myJobTypes.getRandomElement();
23     myProcessPlan = myType.mySequence.getIterator();
24     setName(myType.myName);
25   }
26
27   public void doNextJobStep(){
28     initialize();
29     if (myProcessPlan.hasNext()){
30       JobStep step = myProcessPlan.next();
31       setTimeRequested(step.getProcessingTime());
32       WorkStation w = step.getWorkStation();
33       w.arrive(this);
34     } else {
35       myType.mySystemTime.setValue(getTimeSinceCreation());
36     }
37   }
38 }
```

Accepted Pending Final Revisions

Code Listing 6: The WorkStation Class in the Job Shop Model

```
1  public class WorkStation extends SchedulingElement {
2
3    private Queue myQueue;
4    private Resource myResource;
5    private EndServiceListener myEndServiceListener;
6
7    public WorkStation(ModelElement parent){
8      this(parent, 1, null);
9    }
10
11    public WorkStation(ModelElement parent, int numServers, String
   name) {
12     super(parent, name);
13     myQueue = new Queue(this, getName() + "_Q");
14     myResource = new Resource(this, numServers, getName() + "_R");
15     myEndServiceListener = new EndServiceListener();
16   }
17
18   public void arrive(JobGenerator.Job job){
19     myQueue.enqueue(job);
20     if (myResource.isIdle()){
21     Request nextJob = (Request)myQueue.peekNext();
22     if (job == nextJob){
23       myQueue.removeNext();
24       myResource.allocate(job);
25         scheduleEndService(job);
26     }
27    }
28   }
29
30   private void scheduleEndService(JobGenerator.Job job) {
31     double t = job.getTimeRequested();
32     scheduleEvent(myEndServiceListener, t,
33         "Job " + job.getId() + " End Service at " + getName(), job);
34   }
35
36   class EndServiceListener implements ActionListenerIfc {
37     public void action(JSLEvent event) {
38       JobGenerator.Job job = (JobGenerator.Job)event.getMessage();
39       myResource.release(job);
40       if (myQueue.size() > 0 ) {// queue has a job
41       JobGenerator.Job nextJob =
   (JobGenerator.Job)myQueue.removeNext();
42     myResource.allocate(nextJob);
43         scheduleEndService(nextJob);
44       }
45       job.doNextJobStep();
46     }
47   }
48 }
```

Accepted Pending Final Revisions

Table 1: Sample Output for Job Shop Model

	JSL Event View		Arena	
	Avg	**HW**	**Avg**	**HW**
Waiting time in Q1	3.415	0.18	3.539	0.21
Waiting time in Q2	31.968	4.36	31.650	5.42
Waiting time in Q3	0.188	0.00	0.186	0.00
Waiting time in Q4	9.014	0.81	8.894	0.73
Waiting time in Q5	0.965	0.02	0.961	0.02
Number in Q1	13.688	0.75	14.164	0.87
Number in Q2	64.157	8.85	63.675	11.07
Number in Q3	0.752	0.01	0.746	0.01
Number in Q4	25.286	2.29	24.916	2.05
Number in Q5	1.929	0.04	1.92	0.03
Number Busy at W1	2.883	0.00	2.881	0.00
Number Busy at W2	1.979	0.00	1.978	0.00
Number Busy at W3	2.902	0.00	2.900	0.00
Number Busy at W4	2.922	0.00	2.920	0.01
Number Busy at W5	0.799	0.00	0.798	0.00
System Time Type 1	39.055	4.42	38.874	5.44
System Time Type 2	15.264	0.83	15.262	0.81
System Time Type 3	49.496	4.51	49.137	5.54

Accepted Pending Final Revisions

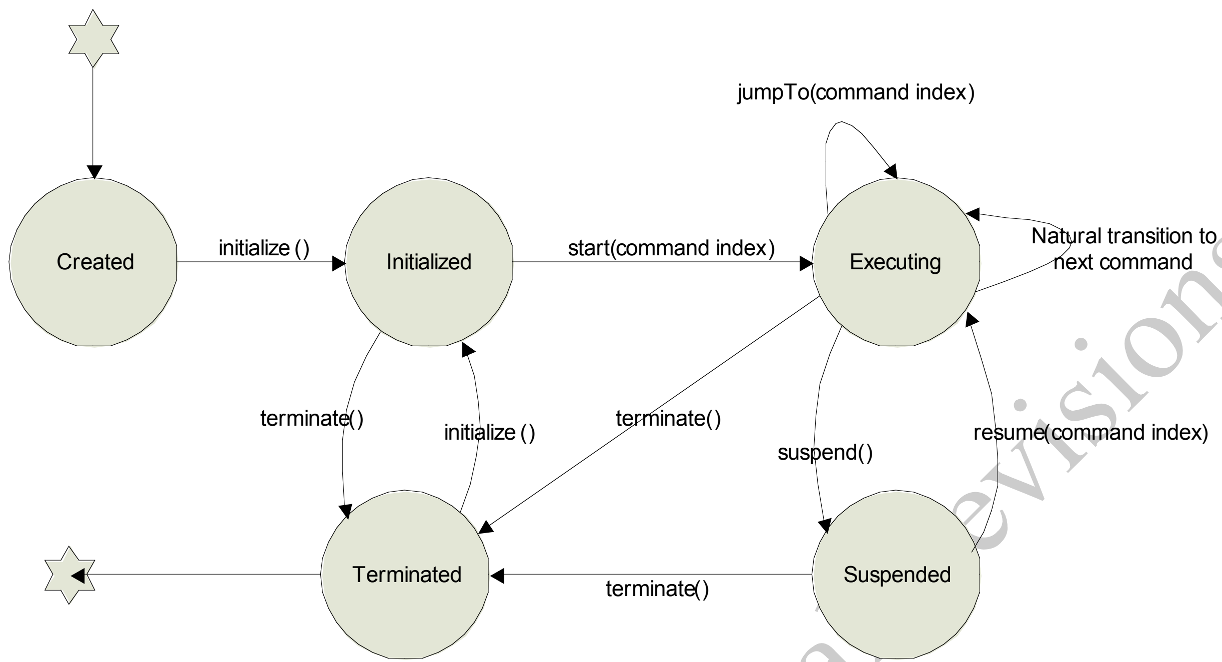

Figure 5: Process State Transition Diagram

Accepted Pending Final Revisions

Code Listing 7: Job Shop Model with Process View

```
1  // create the containing model
2  Model m = Model.createModel();
3  // create the elements in the model
4  Variable v1 = new Variable(m, 1.0, "amt requested");
5  RandomIfc s2w4 = new Gamma(2.0, 1.1/2.0);
6  RandomVariable sts2w4 = new RandomVariable(m,s2w4,"ST S2 W4");
7  RandomIfc s2w1 = new Gamma(2.0, 0.8/2.0);
8  RandomVariable sts2w1 = new RandomVariable(m,s2w1,"ST S2 W1");
9  RandomIfc s2w3 = new new Gamma(2.0, 0.75/2.0);
10 RandomVariable sts2w3 = new RandomVariable(m,s2w3,"ST S2 W3");
11 // create the resources and queues for each workstation
12 Resource r1 = new Resource(m, 3, "W1");
13 Queue q1 = new Queue(m, "W1 Queue");
14 Resource r3 = new Resource(m, 4, "W3");
15 Queue q3 = new Queue(m, "W3 Queue");
16 Resource r4 = new Resource(m, 3, "W4");
17 Queue q4 = new Queue(m, "W4 Queue");
18 // create the process description
19 ProcessDescription jt2 = new ProcessDescription(m,"Job Type 2");
20 // create the time btw arrivals
21 DistributionIfc tbajt2 = new Exponential((10.0/5.0)*0.25);
22 // create the model element to generate type 2 jobs
23 new EntityProcessGenerator(m, jt2, tbajt2, tbajt2);
24 // create the commands used in the process description
25 // workstation 4
26 jt2.addProcessCommand(new Seize(m, v1, r4, q4));
27 jt2.addProcessCommand(new Delay(m, sts2w4));
28 jt2.addProcessCommand(new Release(m, r4, q4));
29 // workstation 1
30 jt2.addProcessCommand(new Seize(m, v1, r1, q1));
31 jt2.addProcessCommand(new Delay(m, sts2w1));
32 jt2.addProcessCommand(new Release(m, r1, q1));
33 // workstation 3
34 jt2.addProcessCommand(new Seize(m, v1, r3, q3));
35 jt2.addProcessCommand(new Delay(m, sts2w3));
36 jt2.addProcessCommand(new Release(m, r3, q3));
```

Accepted Pending Final Revisions