Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

COMPONENT-BASED SIMULATION MODELING

Arnold H. Buss

Operations Research Department
Naval Postgraduate School
Monterey, CA 93943-5000, U.S.A.

ABSTRACT

This paper presents a component-based framework for designing simulation models and discusses its implementation in a package called Simkit. In this framework, Components are defined to be monolith software entities that interact with other components in one of only three ways. Although seemingly restrictive, this approach supports more extensibility and customization of simulation models than conventional Object-Oriented design.

1 INTRODUCTION

This paper presents a component-based framework that is useful for designing and implementing discrete event simulation (DES) models. This paper is an attempt to solidify the grounds for component-based simulation modeling. The framework consists of a few simple elements that enable a considerable amount of flexibility in creating simulation models. It will be illustrated with a package called Simkit.

Component-based simulation modeling is complementary to OO modeling. While the approach we take here is implemented in an object-oriented language (Java), it is possible to implement it in a non-OO system, such as Microsoft's COM.

Components appear to be the ideal level of granularity for implementing simulation models. Most commercial simulation software packages provide a Graphical User Interface (GUI) with the ability to choose relatively large parts of the model from a palette. For example, a standard factory simulation package will contain palette items for workstations, material buffers, and workers. The success of these commercial packages may give the impression that component frameworks are ubiquitous and that the fundamental problems of Component-based simulation modeling have been solved.

Unfortunately, there are many difficulties with the standard approach, and the resemblance of the GUIs to a true component-based framework is superficial. The evidence for this statement lies in the fact that every commercial simulation product relies critically on low-level external code to customize their entities beyond the original design. This external code is written in a language such as Fortran, C, or Visual Basic. A true Component-based framework would not require the modeler to work outside basic components this way.

The remainder of this paper is organized as follows. In the following section we will discuss the basics of Component-Based Design in general terms. Section 3 gives a brief overview of Design Patterns, a critical aspect of software design that is particularly useful for components. Section 4 then gives the basic structure of our approach to component-based simulation modeling. Section 5 discusses the Listener pattern, a particularly important design pattern that greatly facilitates loose coupling of components, and concludes with two useful applications of the pattern. Section 6 briefly touches on some other useful design patterns, and finally Section 7 is a short discussion and presents conclusions.

2 COMPONENT BASED DESIGN

Component-based simulation modeling differs from OO modeling in several ways. In OO design, inheritance and overloading are the primary—indeed, the only—mechanisms for implementing polymorphism. Component-based design prefers to utilize an approach based on establishing common interfaces between components rather than exclusive use of inheritance. Therefore, the primary design tasks for Component-based design are mapping the tasks to be performed to corresponding software components and deciding how the components are to communicate.

A key difference between simply designing with components and Component-Based Design is the degree of coupling necessary between the components. Coupling here is defined to be the extent to which one component must be aware of another when interacting. In this paper we will describe several design patterns that have proved to be useful in implementing Component-Based Design for Discrete Event Simulation modeling.

A framework for Component-Based Modeling may or may not be Object-Oriented. Sun's JavaBeans is an example of an object-oriented framework, while Microsoft's COM is a non-object-oriented component framework (while COM may be *implemented* in an OO language, COM itself is not Object-Oriented). The critical features that make a design Component-Based have to do with the extent to which the components are monolithic and the degree of loose coupling between the components. The approach we take here is closer to JavaBeans in implementation but has elements present in COM, particularly with its reliance on the use of interfaces.

Although a satisfactory definition of component remains elusive, we will use the following one in this paper. A *component* is a monolithic programming entity whose external interface consists only of *property accessor/mutator methods,* of *action methods*, and *event handler methods*.

Property accessor/mutator methods are small methods whose only purpose is to enable reading/writing a single property. Commonly used synonyms are "setters" and "getters." In fact, the JavaBeans component framework prescribes that all properties be accessed only through set<Property> and get<Property> methods.

An *event handler method* is a method that supports the Listener pattern (discussed below in Section 5). It should usually be part of an interface for which it is the only prescribed method. Its signature is always the event of interest. For example, for the SimEventListener pattern discussed in Section 5.1 the method has the signature `processSimEvent(SimEvent)` and is specified in the `SimEventListener` interface.

An *action* method changes the state of the component in ways that are typically more complicated than simply setting the value of a property. In Simkit's implementation there are two kinds of action methods: those built into Simkit for interaction with the Event List and user-defined Event methods that are invoked whenever the appropriate event occurs. The most important Simkit-defined action methods are `waitDelay()` and `interrupt()`. The `waitDelay()` method schedules a SimEvent—that is, places a SimEvent on the Event List—and the `interrupt()` method removes a previously-scheduled SimEvent from the Event List. The only user-defined action methods are those identified by the SimEvent through the original scheduling `waitDelay()` method. The most often used signature is `waitDelay(String, double)`, in which the first argument is the name of the Event method and the second is the amount of simulated time to elapse before the event occurs. In Simkit the Event method has the same name as the Event but with the String "do" prepended. Thus, the Event called "Arrival" has corresponding Event method "`doArrival()`" and is scheduled by the invocation "`waitDelay("Arrival", delay)`."

A related set of methods are associated with registering and unregistering Event Listeners and with dispatching the Event to all registered listeners. These methods comprise the Listener Pattern, which is described in Section 5.

The term "Monolithic" has a negative connotation these days, mostly because it has been applied to legacy systems that have been deemed to be too inflexible and "stove piped." A monolithic system is, by definition, resistant to easy modification and is difficult to extend. Therefore, monolithic is bad in a system. On the other hand, being monolithic is actually a highly desirable property of a component. In Component-Based Design the Component is the fundamental design unit. While the components themselves will be written in a language capable of fine-grained control, a true Component-Based Design eschews this control in favor of exclusive use of Components and their interfaces. Of course, this last statement is an ideal that is hardly ever realized; nevertheless, it is one to which Component-Based models aspire.

3 DESIGN PATTERNS

The concept of software design patterns (Gamma, Helm, Johnson, and Vlissides 1995) provides an extremely useful framework with which to discuss component-based design. A design pattern describes a common, generic solution to a large class of related problems. The seminal book by Gamma *et al* (1995) presents 23 of the most basic design patterns encountered in Object-Oriented software design. Many other useful design patterns have also been discovered.

We will focus primarily on the Listener pattern, one in which components signal their change of state by multicasting events to other objects who have indicated their interest by registration. The Listener pattern can be seen as a lightweight version of the Observer pattern (Gamma, *et al* 1995, p.293). As we discuss below, the Listener pattern is very important for creating component models that are loosely-coupled. Briefly, the Listener pattern's use of event multicasting without requiring callbacks enables components to communicate generically in ways that are self-describing. One variant that is particularly useful is the SimEventListener pattern, described in Section 5 below, in which the method that is ultimately invoked is determined dynamically from the event that is multicast.

In the following section we will briefly describe the component structure that will provide the context for our component-based design. For specificity, we will restrict ourselves to the Simkit implementation; however, the ideas are not restricted to this one implementation, but could be applied in a wider context.

4 BASIC STRUCTURE

Simkit is a package that can aid in implementing component-based simulation models. It is written in the Java^{TM} programming language, which has the added benefit of providing platform-independence as well as network-awareness (Buss and Stork 1996).

Simkit uses Event Graphs (Schruben 1983, 1992; Schruben and Yücesan 1993) as the underlying methodology because of its power and simplicity. Event Graphs provide an expressive and flexible modeling methodology that is very conducive to Component-Based simulation modeling. Event Graphs have just three elements: the Event node, the scheduling edge, and the canceling edge. In Simkit these correspond, respectively, to instance methods having a special prefix ("do" methods), an instance method called `waitDelay()`, and an instance method called `interrupt()`.

Simkit's implementation is broken down into a collection of related interfaces, the most important of which are listed below.

- SimEvent—Events that are scheduled on the event list and subsequently multicast to SimEventListeners.
- SimEventSource—The ability to register listeners and multicast SimEvents.
- SimEventListener—The ability to register and listen to SimEventSources. This interface consists of a single method, processSimEvent(SimEvent).
- SimEntity—The ability to schedule events on the Event List. This is a sub-interface of SimEventSource and SimEventListener. This interface contains the `waitDelay()` and `interrupt()` methods.

SimEvents are placed on the Event list by a SimEntity object by invoking its instance method `wait-Delay(String, double)`. The first argument is the name of the SimEvent, and it is meant to match a corresponding user-written instance method in the SimEntity that gets invoked when the EventList processes the SimEvent. The second argument is the amount of time until the event is scheduled to occur, the delay time. An event occurs when it becomes the next event on the Event List. When that happens, the Event List removes the event and makes a callback to the SimEntity that originally scheduled the SimEvent by invoking its `handleSimEvent(SimEvent)` method. This approach is a simple and typical implementation of an Event List (see Law and Kelton, 2000).

Simkit uses Java's reflection to match the name of an event with a method in the SimEntity that has the same name with "do" prefixed. Thus, the SimEvent called "Arrival" will cause an instance method called doArrival() to be invoked on the scheduling SimEntity when that SimEvent occurs.

After the SimEvent's owner is finished executing the corresponding method, it dispatches the SimEvent to all its registered SimEventListeners (see Section 5.1). This Listener pattern is a key feature of component-based modeling, and we will now discuss it in more detail.

5 THE LISTENER PATTERN

The Listener Pattern provides the primary mechanism by which simulation components communicate. Two types of components are involved with a listener pattern: the listener component and the Event Source component. The "listening" component registers interest in another component's events and waits for the other component to fire the event. When the event fires in the simulation component, it notifies all its registered listeners of the event. Note that the term "event" here is distinct from the simulation events that come off the event list. No matter how many of these events are fired, no simulated time passes. Indeed, the firing of these events can technically be considered to be part of the state transition function for the current simulation Event.

Three entities are involved with every implementation of the Listener pattern: The Event, the Listener, and the Event Source. The same component can serve as a Listener to some components and be an Event Source to other components. The Event that is fired should contain enough information for the Listener to be able to decide what to do *without a callback to the Event Source*. This no-callback property is a critical one for maximizing the looseness of the coupling between components since such a callback requires the listener to have knowledge of the event source object. Indeed, this feature distinguishes the Listener pattern from the Observer pattern (see Gamma, Helm, Johnson, and Vlissides, 1995), since the latter typically does require a callback to the event source.

For maximum flexibility the Listener should be implemented as an interface consisting of just the single notification method with a signature consisting of a reference to the dispatched event.

The Event Source component has three tasks: to register Listener components, to unregister Listener components, and to fire the Event at the proper time. The Event Source is particularly amenable to the use of delegation to perform its task.

Note that the use of an interface to implement the Listener pattern is critical to its extensibility. Implementing a Listener as a class, whether concrete or abstract, restricts all further Listeners to be subclasses. In fact, there is an Interface design pattern that is appropriate here (Gamma, Helm, Johnson, and Vlissides, 1995). The Interface pattern is easily implemented using a Java interface, enabling disparate classes without any "is-a" relationship whatsoever to be first-class participants as Listeners.

The power of the Listener pattern stems from the fact that the Event dispatching can be implemented generically, with the Event Source having to know only that the receiving component implements the Listener interface.

The interface for a Listener typically consists of a single method with one argument, a reference to the dispatched Event. The event source uses this method to make a callback to each listener when the Event is dispatched. Thus, the interface for the event source consists (at a minimum) of methods for registering and unregistering Listeners and at least one method to trigger an Event dispatch.

We will now discuss two uses of the Listener pattern that have proved very useful for Discrete Event Simulation Modeling: The SimEventListener and the PropertyChangeListener. These will be presented as they are implemented in Simkit.

5.1 SimEventListener

The SimEventListener pattern involves an event that has been executed by the Event List. It consists of the source of the event (the SimEntity that scheduled it) multicasting the same SimEvent to registered SimEventListeners.

The callback method from the Event List for a SimEntity is `handleSimEvent(SimEvent)`, which simply invokes the `processSimEvent(SimEvent)` method defined by SimEventListener. The SimEvent contains data (in the form of a String) about which method is to be invoked and optionally a parameter list (in the form of an array of Objects) to be passed to the method. Java's reflection mechanism is used to find the desired method and to invoke it. The invoked method is determined by prepending "do" to the event name and matching a method of that name with a signature consistent with the parameter list. When processSimEvent() returns, then notifyListeners(SimEvent) is called, thus multicasting the SimEvent to all registered SimEventListeners. The SimEventListener interface defines just the `processSimEvent(SimEvent)` method Thus making it very easy for components to define different ways to respond to SimEvents. For example, instead of the slower (but flexible) reflection used by SimEntityBase, Simkit's default SimEntity base class, the desired method could be invoked using a static switch statement based on magic numbers. Another example occurs when a base class that is not a SimEventListener has already been identified. The class has only to declare that it implements SimEventListener and then actually implement the `processSimEvent(SimEvent)` method. This is typical of the way Java implements polymorphism and is an alternative to the more typical use of multiple inheritance.

A SimEntity can only multicast a SimEvent it has previously scheduled; a "heard" SimEvent is not re-multicast. This enables two SimEntities having the same Event to listen to each other without generating an infinite loop. Of course, it is always possible to programmatically create cycles of scheduled events, but each new event must be explicitly scheduled.

5.2 PropertyChangeListener

The PropertyChangeListener pattern specifically involves components changing a property value and notifying interested listeners about that change. The Java language provides support for this pattern with the PropertyChangeEvent and the PropertyChangeListener interface, part of the "JavaBeans" conventions. A PropertyChangeEvent instance contains the property's name, references to both the old and new values, and a reference to the source of the PropertyChangeEvent to support callbacks.

The PropertyChangeListener pattern is useful in simulation models for handling state variables and their changes and Simkit adopts the convention of firing PropertyChangeEvents whenever state variables change value.

The PropertyChangeListener interface has a single callback method, propertyChanged(PropertyChangeEvent) that is invoked when a property is fired. The PropertyChangeSupport class has methods for registering and unregistering PropertyChangeListeners and for firing PropertyChangeEvents. An object can delegate the management of the PropertyChangeListener pattern to an instance of PropertyChangeSupport.

The PropertyChangeListener pattern is more useful than a SimEventListener when the listening component is primarily interested in the state changes rather than the occurrence of a particular event. The property itself could in fact be present in more than one simulation component; and a PropertyChangeListener could be registered with all components managing a particular property. Furthermore, a component only concerned with the state variable would have to make a callback to the source if it used the SimEventListener pattern to hear the property changes. A PropertyChangeEvent, in contrast, contains all the necessary state information for that variable.

The SimEventListener pattern is more useful when the occurrence of the event is the important piece of information to the listening component rather than the state variables. In the queueing example in Section 5.3 below, the Server component is listening to the ArrivalProcess component for the Arrival event and does not care about the state of the ArrivalProcess.

5.3 Example

In this section we will illustrate the SimEventListener pattern by showing how a simple Arrival Process SimEntity can be defined to create Arrival events that are listened to by a Server SimEntity in a loosely coupled way. We will use Event Graph methodology to describe the simulation components (see Schruben, 1983; 1992; Schruben and Yücesan, 1993) as well as snippets of Simkit code.

5.3.1 The ArrivalProcess SimEntity

The Arrival Process is the simplest non-trivial DES model. It consists of a single event, Arrival, whose occurrence triggers another Arrival event with a delay given by a stream of interarrival times $\{t_A\}$ that may be deterministic or stochastic. Typically a state variable counts the number of events that have occurred. The ArrivalProcess can thus be thought of as a renewal process, although it is slightly more general since it can support a correlated stream of interarrival times. The Event Graph for the ArrivalProcess is shown in Figure 1.

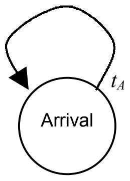

Figure 1: The Arrival Process

The Simkit code snippet corresponding to the Arrival Process Event Graph of Figure 1 is as follows:

```
public void doArrival() {
  waitDelay("Arrival",
          interarrivalTimes.generate());
  }
```

The waitDelay() method schedules the Arrival event after a delay given by the second argument, which in this case is randomly generated. When an Arrival event is processed from the Event List, the doArrival() method is invoked, prompting another Arrival event to be scheduled. While the Arrival Process is a complete DES, it holds little interest alone. In the next section we will show how it can be used to stimulate arrivals to a simple queue.

The ArrivalProcess is implemented as a simulation component in Simkit using the above code in a Java class. The ArrivalProcess component is initialized by configuring the random interarrival time generator and by scheduling the first Arrival event. Once it is verified to be correct, it need not be modified for use with a second component that models the server portion of a queueing model.

5.3.2 The Server SimEntity

The Server SimEntity models a multiple-server queue in which arrivals wait in a queue until they can be served by one of several servers (see, for example, Law and Kelton; 2000). The Server can be modeled with two state variables, `numberInQueue` and `numberAvailable Servers`, and three events: Arrival, StartService, and EndService. The Server Event Graph is shown in Figure 2. The Simkit methods for the Server SimEntity are shown below:

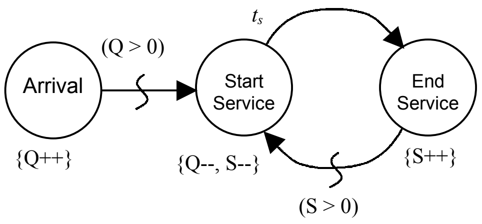

Figure 2: The Server Event Graph

```
public void doArrival() {
    firePropertyChange("numberInQueue",
        numberInQueue,
        ++numberInQueue);
    if (numberAvailableServers > 0) {
        waitDelay("StartService", 0.0);
    }
}
public void doStartService() {
    firePropertyChange(
        "numberAvailableServers",
        numberAvailableServers,
        --numberAvailableServers);
    firePropertyChange("numberInQueue",
        numberInQueue,
        --numberInQueue);
    waitDelay("EndService",
        serviceTimes.generate());
}
public void doEndService() {
    firePropertyChange(
        "numberAvailableServers",
        numberAvailableServers,
        ++numberAvailableServers);
    if (numberInQueue > 0) {
        waitDelay("StartService", 0.0);
    }
  }
```

Note that we have added the firePropertyChangeEvent() methods in the above code whenever a state variable changed values. In general, a PropertyChangeEvent should always be fired whenever a state variable changes value. Unlike the ArrivalProcess, the Server is not a complete DES, but must be connected with another SimEntity that generates Arrival events.

5.3.3 Connecting ArrivalProcess and Server using the SimEventListener Pattern

To create a complete DES for the queueing model the ArrivalProcess and Server SimEntities must be connected by the SimEventListener pattern. Specifically, an instance of Server must be made a SimEventListener to at least one instance of a SimEntity that generates Arrival events. This is accomplished by the following Simkit code:

```
SimEntity arrivals = new ArrivalProcess(...);
SimEntity server = new Server(...);
    arrivals.addSimEventListener(server);
```

The above code first creates the two SimEntities, then adds the server instance as a SimEventListener to the arrivals instance. Now whenever the Arrival event occurs in the arrivals instance, an Arrival event will also be triggered in the server instance; that is, the doArrival() method of server will be invoked. No additional Arrival events will be put on the Event List as a result of this listening; conceptually, the Arrival event in the server is just part of the overall Arrival event.

The code necessary to implement the SimEventListener pattern is implemented in a SimEntityBase class. The generic implementation is such that the modeler does not need to configure the Event listener. Rather, the SimEvent carries the Event name that matches an instance method with the string "do" prepended (see the code above). Any arguments to the method are also referenced by the SimEvent and passed to the "do" method when it is invoked.

Connecting SimEntity objects with the SimEventListener pattern enables components to be loosely-coupled—they can interact in meaningful ways simply by being connected. The listener pattern provides an extremely flexible way to connect simulation components. For example, an number of SimEntity objects can provide the Arrival event necessary to stimulate an arrival to the Server process described above. Furthermore, any SimEntity having the Arrival process (with zero arguments) can provide the stimulus, so long as the Server instance is added as a SimEventListener to it.

The SimEventListener pattern also enables reuse of components by enabling them to be connected in new ways using very small "adapter" classes. The simplest version of an adapter class involves listening to one event in a component and scheduling another event in another component. For example, the simple queueing model above consists of two components, an ArrivalProcess and a Server. Suppose that the modeler wishes to use these two classes to create a transfer line consisting of n multiple-server queues in series. It is easy enough to create n Server instances, but they cannot be directly connected by the SimEventListener pattern (recall that all events are heard, so a StartService event in one Server instance will stimulate a StartService event in the next, not the desired behavior). Instead, a small SimEntity called MaterialHandler is created that listens to the EndService event and schedules the Arrival event, as shown below:

```
public class MaterialHandler
                    extends SimEntityBase {
    public void doEndService() {
        waitDelay("Arrival", 0.0);
    }
    }
```

To create the transfer line, the modeler must simply instantiate one instance of MaterialHandler after each Server instance (except, of course, for the last one)

5.4 Statistics Gathering using the Listener Pattern

A loosely coupled approach to gathering statistics is supported by the PropertyChangeListener pattern. All state variables in Simkit should fire a PropertyChangeEvent when their value is changed. Any components designed to gather information about a simulation can obtain all state trajectories by implementing the PropertyChangeListener interface. Unlike the SimEventListener pattern, the PropertyChangeListener pattern is more ingrained in Java. Indeed, PropertyChangeEvents are part of the standard Java 2 class files. This means that many components that were not written with Simkit in mind may nevertheless be registered as PropertyChangeListeners to SimEntity objects and received state changes.

One example of using a PropertyChangeListener for statistics is Simkit's SimpleStats class. This class is a standard statistical accumulator class. As values are added, an instance of SimpleStats updates internal accumulators for the count, sum, sum of squares, minumum, and maximum values. When SimpleStats is instantiated with the name of a property then whenever it hears a PropertyChange event with the name of that property, it uses the new value of the PropertyChangeEvent to update its accumulators. Thus, a state variable could be changed in many different components, yet the data be properly recorded by ensuring that a SimpleStats with that property name was listening to each object in which the state variable was changed.

The PropertyChangelistener pattern is also effective for displaying state variables graphically. A PropertyChangeListener that updates its graph is all that is required. Like the SimpleStats example above, the property could exist in many different objects.

One advantage of using the PropertyChangeListener pattern is that the state variable (property) names can be decoupled from the actual names of the instance variables, thus hiding the inner details of the implementation. This is useful if state variable name conflicts need to be resolved or if the property being fired is some function of state variables. For example, suppose that internally the Server class kept a state variable called NumberAvailableServers that modeled the number of available servers at any given moment. However, suppose the designer of the class wished to represent the number of busy servers as the state variable. This could be done by simply firing a PropertyChangeEvent called "NumberBusyServers" that was the difference between the total number of servers and the number of available servers.

A significant modeling benefit to the PropertyChangeListener pattern is that simulation components can be written without any statistics gathering code in them, as long as PropertyChangeEvents are fired for each state change. This not only reduces the size of model code, but it imparts a great deal of flexibility. The modeler can write simulation classes with the only consideration being the correct sequence and timing of state transition in the model. The code will never have to be subsequently revised simply because some state variable was not collected.

6 OTHER DESIGN PATTERNS

There are more design patterns that are very useful for supporting Component-Based simulation modeling. One particularly important one is the Abstract Factory Pattern (Gamma, *et al* 1995, p.87). Instead of invoking a constructor, an Abstract Factory is called to request an instance of an object implementing a desired interface. The Abstract Factory instantiates the object and returns it as a reference typed to the interface. This is critical to maintaining a clean separation between interface and implementation.

Simkit uses an Abstract Factory to separate random number generation and random variate generation while making either one extremely configurable. For example, in a Simkit model it is possible to change the probability distribution without recompiling the model. This change can even be made on a running program.

Another useful pattern is the Mediator pattern (Gamma *et al*, p, 273). This pattern is responsible for adjudicating interactions between two simulation components. It has been applied in military simulation models to capture interactions between a sensor and a potential target. A Mediator component contains the particular detection algorithm used by the sensor to detect the target. The pattern's use enables different detection algorithms to be easily used in a model. The Mediator pattern furthermore keeps private data regarding the ground truth of the target away from the sensor component (and vice versa). It is important to separate this information so it is not inadvertently used by the sensor to improperly detect or locate the target.

A related pattern is the Referee, which is responsible for determining which components will in fact interact. Its responsibility typically involves nothing more than assigning the proper Mediator to two components at the appropriate time. It therefore only has to determine the existence of the interactions and then delegate the "work" to the correct Mediator.

7 DISCUSSION AND CONCLUSIONS

At first glance, the loosely coupled component modeling described here may appear similar to many commercial discrete event simulation packages. "Drag and Drop" visual modeling has the feel of a component-based framework. The main difference between these packages and the component approach taken in this paper is the tightness of the coupling between the components. In this framework, components are loosely coupled by the two listener patterns. Any simulation component may listen to any other one and likewise any PropertyChangeListener component can hear PropertyChangeEvents from any component firing those events. In contrast, the components found in the visual scenario builders are restricted in what they communicate with and relatively inflexible about the nature of that communication.

The component-based simulation framework presented here is more flexible and extensible than that of traditional OO modeling alone. The transmittal of messages is done using the two generic Listener patterns described above rather than maintaining an explicit reference to the recipient of the message (see, for example, Joines and Roberts 1999). The two types of messages in our framework have proved to be sufficient for all simulation modeling purposes encountered by the author.

Design patterns have proved to be an extremely powerful tool for software modeling and the component framework described in this paper has explored the use of a few such patterns. There is great potential for more extensive use of design patterns in simulation software design.

The component-based framework presented in this paper can be implemented in a number of languages. Any language that supports delegation, interfaces, and runtime type information in addition to traditional object-oriented features is a viable candidate. There are some features of the Java™ programming language that have proved to be particularly enabling. Two in particular are the language-level support for interfaces and the language's support for reflection. Interfaces enable a cleaner separation between the public "view" of a component and the implementation details than classic OO design. Specifically, it is possible for classes with no "is-a" relationship whatsoever to be interchangeable providing they implement a common interface. Interfaces also are key to the use of Abstract Factories to obtain instances. Reflection allows objects to reveal information about themselves through Class and Method objects, thus making it possible to write extremely generic code to "discover" the structure of a simulation component.

ACKNOWLEDGMENTS

The author would like to thank Lee Schruben for Event Graphs and Gordon Bradley for comments on an earlier version of the paper. He would like to especially thank Paul Sanchez for his discerning critical feedback as well as for many useful discussions about software design. This research was supported by a grant from the Air Force

Office of Scientific Research. This support is gratefully acknowledged.

REFERENCES

Buss, A. and K. Stork. 1996. Simulation on the world wide web using Java. In *Proceedings of the 1996 Winter Simulation Conference*, ed., J. Charnes, D. Morrice, D. Brunner, and J. Swain. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Joines, J. and Roberts, S. 1999. Simulation in an object-oriented world. In *Proceedings of the 1999 Winter Simulation Conference*, ed., P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans,. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Law, A. and D. Kelton. 2000. *Simulation Modeling and Analysis, Third Edition*, McGraw-Hill., Boston, MA.

Schruben, L. 1983. Simulation modeling with event graphs. *Communications of the ACM* 26: 957-963.

Schruben, L. 1992. *Sigma: A Graphical Simulation Modeling Program*. The Scientific Press. San Francisco, CA.

Schruben, L and E. Yücesan. 1993. Modeling paradigms for discrete event simulation. *Operations Research Letters.* 13: 265-275.

Stork, K. 1996. *Sensors In Object Oriented Discrete Event Simulation*. Masters Thesis, Operations Research Department, Naval Postgraduate School, Monterey, CA.

AUTHOR BIOGRAPHY

ARNOLD H. BUSS is an Assistant Professor in the Operations Research Department at the Naval Postgraduate School. He received a B.A. in Psychology from Rutgers University, his M.S. in Systems Engineering from the University of Arizona, and a Ph.D. in Operations Research from Cornell University. His recent work has involved Component-Based software design. His email address is `<bussa@or.nps.navy.mil>`.