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

SILK, JAVA AND OBJECT-ORIENTED SIMULATION

Richard A. Kilgore

ThreadTec, Inc
P. O. Box 7
Chesterfield, MO 63006, U.S.A.

ABSTRACT

Silk® is a set of Java classes that support object-oriented, general-purpose simulation and animation using the Java programming language. Silk enables the development of complex, yet manageable simulations through the construction of usable and reusable simulation objects. Silk objects are usable because they express the precise behavior of individual entity-threads from the object perspective using familiar process-oriented modeling constructs and the object-oriented features of a general purpose programming language. Silk objects are reusable because they can be easily archived, edited and assembled using professional Java visual development environments that support the JavaBeans component architecture. This introduction describes the fundamentals of designing and creating a Silk model.

1 INTRODUCTION

The Silk language was designed to be different things to different people. To some users, Silk is a set of basic simulation Java class libraries that can be creatively assembled into a variety of new modeling constructs. To others, Silk is a process-oriented modeling language that offers the power and flexibility of a standard programming language. To others, Silk is a visual modeling environment where Silk-based modeling components can be graphically assembled to quickly create simulation applications. Most importantly, Silk is the first practical implementation of a tool for building object-oriented simulation components and domain-specific simulators.

Silk is not so much a simulation language in itself as it is a simulation extension of the Java language. The power, flexibility and extensibility of Silk derive directly from Java and the clean way in which the simulation-related extensions that constitute Silk have been integrated into the language. Achieving such a high level of integration would not be possible were it not for a unique combination of features in the Java language. One is a simple yet powerful framework that greatly facilitates the implementation of object-oriented design methodology and its capabilities for creating flexible, modular, and reusable programs. Another is Java's built-in support for multi-threaded execution, which is essential to representing in a natural way the flow of entities in process simulation models. By incorporating the Silk modeling environment into Java, users also have direct access to Java's native support for browser-based execution, standard Internet communication protocols, database connectivity and graphical user interface development. Java's platform neutral design also means that Silk models can be developed and executed on practically any combination of computer hardware and software platforms.

This paper is intended to serve primarily as an introduction to the language level features of Silk which serve as the foundation for developing reusable simulation components and higher level domain-specific simulators. Other articles dealing with other important aspects of the Silk language are listed in the References section. Section 2 contains a overview of creating an object-oriented model with Silk. Section 3 describes the development of Silk in Java Integrated Development Environments. Section 4 is an overview of Silk JavaBean components for visual modeling and animation. Section 5 contains concluding remarks.

2 OBJECT-ORIENTED DESIGN WITH SILK

Silk is the most powerful when the user follows a consistent design pattern for object-oriented modeling in which each "intelligent" component is modeled as an independent Silk entity class.

To illustrate this concept, consider the typical single-server queuing system of a customer served by a bank teller. There are two intelligent components in the system capable of independent thought and action, the Customer and the Teller. This system could be modeled in Silk as a single class from either a pure Customer-push or Teller-pull perspective. But there are substantial design benefits to adherence to the proper object-oriented simulation representation of the system in which there is one Silk simulation class for each intelligent system component. The Silk representation of the Customer and Teller classes are described in Figures 1 and 2. The numbers in brackets in the text will refer to the line numbers in these figures and color is used to distinguish Java keywords (blue), Silk keywords (red), comments (green) and user-defined identifier labels (black).

2.1 The Customer Class

A Silk simulation class is like any typical Java programming class. The packages of Silk classes referenced are identified in import statements [1-3] and the user-defined class name Customer is declared as an extension of the Silk Entity class [4]. The class structure consists of the data declarations [6-13] which will define the character-istics of the simulation entities created from this class and the default Silk *process* method [14-28] that will change those entity characteristics as the state of the system changes. The essence of Silk is the use of these Entity methods, Java statements and other Silk objects within this process method to represent exactly what behavior that the real systems entity experiences.

Each instance of this Customer class is assigned two unique, user-defined attribute indentifiers, *attArrivalTime*, *attServiceDelay* [6]. Since a Silk simulation is a Java program, these attributes can be any Java or Silk data type. In this case, a Java double precision variable is needed, but the service times might be an entire array of process objects that define tasks, resources required and service times (please see www.threadtec.com/models for more industrial-strength examples).

While each Customer instance will have these unique attribute identifiers, all instances of the Customer class will share common *static* class variables representing Java or Silk objects [8-13]. Only Silk objects for random variable generation and statistics are shown in this example, but again remember that Silk models are Java programs so the entire collection of Java data types and objects available. For example, a more complex model might contain an array of all of the required processing delay distributions that this entity might require.

```
1.  import com.threadtec.silk.*;           // Silk general purpose classes
2.  import com.threadtec.silk.random.*;    // Silk random variable generation classes
3.  import com.threadtec.silk.statistics.*;// Silk statistics collection classes

4.  public class Customer extends Entity {

5.    // Attributes (instance variables) unique to each customer
6.    double attArrivalTime, attServiceDelay;

7.    // Silk objects (class variables) common to all customers
8.    static Exponential   expInterArrivalTime = new Exponential( 10.0 ),
9.                         expServiceDelay     = new Exponential( 8.0 );
10.   static Observational obsTimeInSystem     = new Observational( "Time in System" ),
11.                        obsTimeInQueue      = new Observational( "Time in Queue" );
12.   static Queue         queCustomer         = new Queue( "Customer Queue" );
13.   static TimeDependent timQueue            = new TimeDependent( queCustomer.length,"In Queue");

14. public void process( ){

15.   // create next customer arrival and record arrival time
16.   create( expInterArrivalTime.sample( ) );
17.   attArrivalTime = time;

18.   // assign service time for this customer and wait for service
19.   attServiceDelay = expServiceDelay.sample( );
20.   queue( queCustomer );

21.   // queue delay controlled by teller
22.   halt( );                      // suspend process until teller activates
23.   obsTimeInQueue.record( time - attArrivalTime ); // record queue time

24.   // service delay controlled by teller
25.   halt( );                      // suspend process until teller activates
26.   obsTimeInSystem.record( time - attArrivalTime ); // record system time

27.   dispose( );

28.   }// end of process method
29. }// end of Customer class
```

Figure 1: Customer Class Definition

```
1.  import com.threadtec.silk.*;            // Silk general purpose classes
2.  import com.threadtec.silk.statistics.*; // Silk statistics collection classes

3.  public class Teller extends Entity {

4.    static Resource resTeller = new Resource ("Teller");
5.    static TimeDependent timTeller = new TimeDependent( resTeller.numBusy, "Utilization");

6.    public void process ( ) {

7.      while ( true ) {  // Teller not scheduled, continuously seeks new Customers

8.      // wait while condition is true (no customers in queue
9.      while( condition (  Customer.queCustomer.getLength( ) == 0 ) );

10.     // obtain reference to first customer in queue and remove it
11.     Customer entCustomer = (Customer)Customer.queCustomer.remove(1);

12.     // process customer and release teller
13.     seize ( resTeller );
14.     entCustomer.activate( );        // end halt for customer in queue
15.     delay ( entCustomer.attServiceDelay );
16.     entCustomer.activate( );        // end halt for customer in system
17.     release ( resTeller );

18.     }// end of while block for Teller processing

19.   }// end of process method

20. }// end of Teller class
```

Figure 2: Teller Class Definition

A significant advantage of Silk over previous object-oriented languages is the use of process-oriented methods familiar to users of other simulation language. Every Silk class must contain a *process* method containing these statements (or references to other classes that contain these statements) and it is here that the power of object-oriented modeling becomes evident. The *process* method [14-28] describes line for line the sequence of actions and information processing that defines the intelligent behavior of this system component. When the component is waiting for a decision or action of another intelligent component, the entity will halt its process until activated.

In this example, the Customer creates [16] the arrival of the next Customer using a sample from a Silk Exponential random variable object created in the data declaration. The *attArrivalTime* variable is then set to the current value of Silk simulation *time* [17]. The "att" prefix is not required and has no special Silk significance other than to remind the modeler that this is an instance variable unique to this object. Next, the *attServiceDelay* variable is then assigned a sample value from the appropriate service time distribution [19]. More complex models would likely have different distributions for different Customer classes and the use of an attribute for service delay will allow the Teller object access to the required processing time for each Customer instance and type.

This assignment of the service time to an attribute of the Customer object is an important object-oriented design choice. Is the time required for service an attribute of the Customer or should it be defined as a characteristic of the Teller? If different Tellers have different performance characteristics in performing the required service, those factors properly belong in the Teller class definition. But the service requirement is a characteristic of the customer and new customer types (which might inherit from this Customer class) should have the ability to modify the default customer service requirement without modification in Teller classes. Small design choices such as this are crucial to the adherence of a consistent design that will make Silk models easier to reuse.

The Silk Entity queue method then places this Customer instance in a Silk *queue* [20] which is simply an ordered list of Customers. Note that this queue is not linked with any particular Silk *Resource* object so an Entity can be simultaneously listed in any of a number of *Queues.* This is extremely useful for modeling complex server behavior and facilitates proper statistics collection.

Until this point, the Customer entity is an intelligent component that has "pushed" to join the Teller queue. In the actual system, control of the choice of which Customer is served next is now passed to the Teller object. Consequently, the Customer object is halted by a Silk *halt* method [22]. This distinction may seem cumbersome at first and the traditional entity-push approach could be used throughout the Silk process definition as is typical in other process-oriented languages. But the object-oriented design requires that data characteristics and behavior of each object is encapsulated within that object. The significance of this approach will become clearer as the behavior of the Teller object is described.

Once chosen for service, the Customer is activated by the Teller object [shown in Figure 2, line 14]. The Customer object proceeds to record the time spent in the halted state in a Silk Observation statistic object [23]. The TimeDependent object for Customer queue length [13] is automatically updated each time that the queue characteristic *length* is changed. Similarly, the end of service is also under the control of the Teller object so the Customer is again halted [25] until service is completed and the Customer is activated by the Teller object [Figure 2, 16]. Statistics for system time are then recorded for system time [26], and this instance of the Customer class is then disposed [27]. The dispose method actually results in the Silk object being placed in a pool of Customer objects to be reincarnated as representations of future customers.

2.2 The Teller Class

The Silk description of the Teller class is found in Figure 2. It defines the simulation data and behavior from the perspective of the Teller. Since the Teller class is also a system component with independent intelligence, it is a modeled as a Silk Entity [2]. Note that there is a Silk *Resource* object created to represent the Teller state [4]. The responsibility for when and how to change this state from busy to available is left to the *process* method for the Teller [6-19]. The Java *while* block [7] is used to continuously loop the single instance of the Teller throughout the simulation. By default, a Silk entity executes the *process* method only once so this Java construct is necessary to allow the instance of Teller entity to continuously repeat the process method for subsequent Customers.

Interaction between Silk objects is reserved for the most powerful of Silk constructs, the *while(condition( ))* . The *while(condition( )* combines the Java *while* statement and the Silk *condition* method [9]. Similar to the *halt* method, this statement temporarily stops the process of a Silk entity until activated by another process. In this case, the entity proceeds only when the expression defined within the condition method evaluates to false. The user is responsible for stating the conditions for the wait based on the state of Queues, Resources and other Silk or user-defined *state variables*. In this case, the Teller wait can be interpreted as "wait while the length of the Customer queue is 0".

At first look, this structure may appear cumbersome for simple systems. But more experienced modelers will appreciate the ability to create compound conditions for modeling resource behavior based on a variety of factors. Silk performance is unaffected by this complexity as conditions are re-evaluated only when those objects which appear in condition methods change value. Only those conditions where changed state variables appear are evaluated. Note that while many entities may be waiting for the same condition, only one is activated at a time to allow the activated entity an opportunity to change the condition (by seizing a resource or joining a queue).

The net result in the case of the Teller is that the arrival or existence of an entity in the Customer queue results in the continuation of the Teller process. The Teller calls the remove method of the Queue object to obtain a Customer reference and remove the Customer entity from the queue [11]. This statement shows the use of a declaration of an object type within an expression (Customer entCustomer) and also the casting of the object type returned by the remove method to a Customer object type. Silk users commonly "wrap" complex methods like these within other simpler methods user-defined methods of their own creation. But the power of Silk is the ability of users to create and extend the language without sacrificing the underlying power and flexibility of the basic Silk Entity methods.

The Teller object uses the reference to the Customer entity *entCustomer,* to access the service delay attributes of the Customer [15] and to invoke the activate method to resume the process method for the halted Customer entity as described earlier [14,16]. The seize and release methods in [13,17] modify the busy state of the Teller Resource object to allow the TimeDependent object to automatically track Teller utilization [5].

Finally, a brief note is necessary to explain the Silk entity-thread concept enabled by Java. Silk Entities are Java threads. Java's support for multi-threaded execution enables the various types of entity behaviors (halt, delay and while(condition) described above. It is also an essential aspect to the implementation of a natural process-oriented modeling capability in Java. An executive thread running in the background coordinates the management of simulated time and the resumption of suspended threads.

2.3 The SIMULATION Class

All Silk models require a Simulation class, as shown in Figure 3, primarily for the purpose of creating the first instance of each class in the *init* method [3]. The *newEntity* method [5] is responsible for the creation of the Silk object pool of the indicated class and returns a reference to a new or existing member of that pool. The *start* method [6] then begins the execution of the Entity *process* method after a delay of the appropriate time units. In addition, other global parameters may be declared in the Simulation *init* method since all Entities extend Simulation and thus have access to all public variables and methods defined in the Simulation class. Finally, the *run* method of the Simulation class is automatically called by Silk to start the execution of the desired number of runs and run length [12,13]. Execution will end with the creation of a

```
import com.threadtec.silk.*;        // references Silk methods found in this package

public class Simulation extends Silk {

  public void init ( ) {

    // instantiate Silk Entity objects prior to the beginning of run
    Customer entCustomer = (Customer)newEntity( Customer.class ); // create first Customer
    entCustomer.start( 0.0 );

    Teller entTeller = (Teller)newEntity( Teller.class );         // create first Teller
    entTeller.start( 0.0);

  } // end init method

  public void run ( ) {

    // initialize Silk settings and flags prior to beginning of run
    setReplications( 1 );                      // End simulation at the end of 1 replication.
    setRunLength( 10000. );                    // Execute the simulation for 10000 time units
    setControlConsole (true);                  // Use Control Console for interactive control

  } // end run method

} // end Simulation class
```

Figure 3: Simulation Class Definition

Summary Report window or the user can ask that the Silk *Control Console* be used for interactive execution, tracing and animation control [14]. The Simulation class also has a *finish* method that is called at the end of each replication of the simulation to allow programmed execution of complex experimental designs.

2.4 Object-Oriented Design Choices in Silk

As seen in this example Silk provides great flexibility regarding the choice of object-oriented design patterns. Consider the decision to declare the *Queue* object to be a characteristic of the Customer class [12]. Even in this simple example, a Silk user has at least four choices as to the proper assignment of this Queue object. One option if for the Queue object to be declared public and instantiated in the Simulation class which makes the queue reference available in all Silk entity processes. But object-oriented design principles encourage the encapsulation of data and methods in their respective classes so that only those classes which need access to these objects can access these objects. The choice is then between the Teller class, the Customer class or a third class which might contain the physical description of the facility in which the Teller is located.

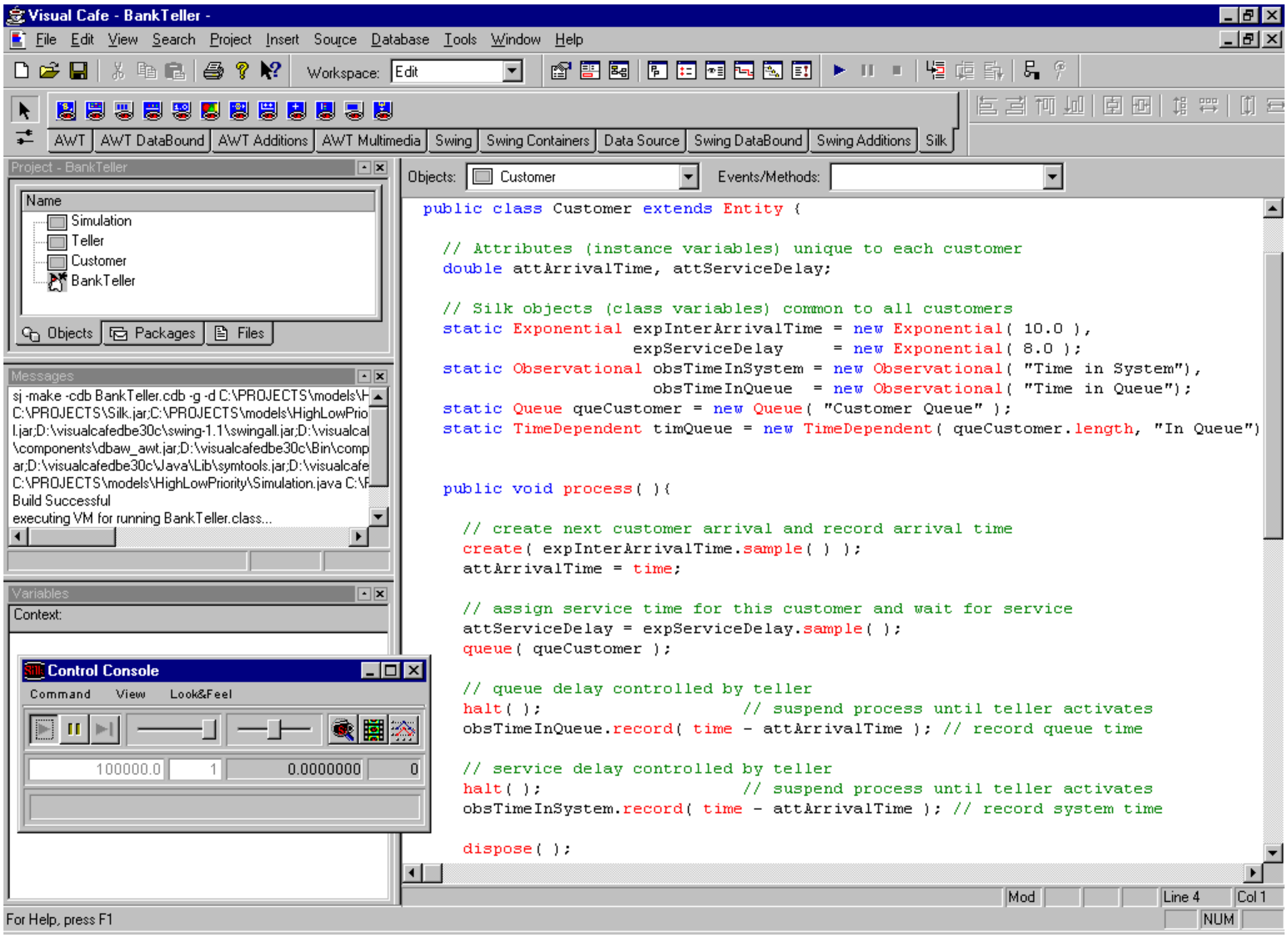
This decision is very important for complex model design and simulation object reusability. Silk modelers are encouraged to create process methods that reflect the actual characteristics and behavior of the corresponding intelligent system component. In this system, the Customer is in control of the behavior regarding which queue to join (and in more complex models, how long to wait in the queue chosen or whether to switch lines, etc.). For that reason, the queue definitions are made in the Customer class so that other versions of the model can change Customer queueing behavior without modifying the Teller class.

3 SILK DEVELOPMENT ENVIRONMENTS

The Silk simulation extensions to the Java language are themselves implemented entirely in Java. The only requirements for building and executing Silk simulation models are a Java language compiler and run-time Virtual Machine that are compatible with Sun's JDK 1.2 specification of the language. Most commercial simulation software vendors constrain users to a single proprietary and often cumbersome development environment. With Silk, users can choose from a variety of professional, third-party Java Integrated Development Environments (IDE's). Each of these IDE's provides a sophisticated graphical interface and a rich collection of tools for project management, source code creation and modification, compilation, debugging, and deployment as standalone applications, browser-based applets, or server-based servlets. Figure 4 contains a screen snapshot of the example problem from the previous section within the Visual Café development environment.

4 JAVABEANS COMPONENT MODELING

Simulation without programming is unrealistic in most industrial strength models and no simulation vendor has the omniscience necessary to create components that do not require modification to represent the subtle, but



```
sj -make -cdb BankTeller.cdb -g -d C:\PROJECTS\models\H
C:\PROJECTS\Silk.jar;C:\PROJECTS\models\HighLowPrio
l.jar;D:\visualcafedbe30c\swing-1.1\swingall.jar;D:\visualcal
\components\dbaw_awt.jar;D:\visualcafedbe30c\Bin\comp
ar;D:\visualcafedbe30c\Java\Lib\symtools.jar;D:\visualcafe
C:\PROJECTS\models\HighLowPriority\Simulation.java C:\F
Build Successful
executing VM for running BankTeller.class...
```

```java
public class Customer extends Entity {

  // Attributes (instance variables) unique to each customer
  double attArrivalTime, attServiceDelay;

  // Silk objects (class variables) common to all customers
  static Exponential expInterArrivalTime = new Exponential( 10.0 ),
                     expServiceDelay     = new Exponential( 8.0 );
  static Observational obsTimeInSystem = new Observational( "Time in System"),
                       obsTimeInQueue  = new Observational( "Time in Queue");
  static Queue queCustomer = new Queue( "Customer Queue" );
  static TimeDependent timQueue = new TimeDependent( queCustomer.length, "In Queue")


  public void process( ){

    // create next customer arrival and record arrival time
    create( expInterArrivalTime.sample( ) );
    attArrivalTime = time;

    // assign service time for this customer and wait for service
    attServiceDelay = expServiceDelay.sample( );
    queue( queCustomer );

    // queue delay controlled by teller
    halt( );                        // suspend process until teller activates
    obsTimeInQueue.record( time - attArrivalTime ); // record queue time

    // service delay controlled by teller
    halt( );                        // suspend process until teller activates
    obsTimeInSystem.record( time - attArrivalTime ); // record system time

    dispose( );
```

Figure 4: Silk Modeling using the Visual Café Integrated Java Development Environment

important differences between system alternatives. But starting a model with Silk components to create a simulation might be preferable to developing applications from scratch. Component-based simulation applications bring economies of speed in development and testing by capitalizing on previous successes. Silk users can leverage the use of JavaBeans as a set of classes and programming conventions that constitute a component development model for the Java language.

JavaBeans are designed to be manipulated graph-ically within visual development environments like Visual Café. The bias is these early years of Java tool design has been toward use of JavaBeans for GUI development where property changes are easier than code changes. But the emergence of Enterprise JavaBeans for cross-platform, cross-developer application development is driving changes in these tools beneficial for simulation-component development. Visual programming allows for the concentration and separation of skills among developers. Skilled programmers build and make available beans for other developers with more domain-specific knowledge (and typically less technical programming expertise) to assemble visually into custom applications. This model works as well for simulation development applications as it does for complex programming applications.

JavaBeans can be applied to any aspect of an simulation application. It is a relatively simple matter to write self-contained, simulation modeling components based on Silk, that automatically make known their functionality and interoperability when incorporated into a JavaBeans visual development environment. Within this environment, they can be added to user-defined component toolboxes or palettes. Users can then assemble components visually into a model by placing them in a workspace and editing their properties to create a desired behavior. None of these manipulations require code to be written by the application developer.

While JavaBeans provides a means for packaging functionality into reusable units; beans by themselves do not ensure reusability. To exploit the potential that simulation components have to offer, policies that define the functionality and modes of interoperability that allow components to be reused must be developed and adhered to. For example, there exist a set of policies and supporting classes in Silk that define the ways in which components must interact with Silk to produce animated displays of system state changes. These conventions were used in the implementation of a core capability in Silk that provides for animating entity movements, entity queueing, entity delays, and numeric and analog displays of state variable values among others. If the prescribed conventions are followed, it is a simple matter for users to modify these existing components or define new ones that will interoperate with any Silk simulation model.

Developing guidelines for enterprise modeling components will be more challenging. Consideration will need to be given to the application domain as well as the range of model granularity the components are required to accommodate. Silk and JavaBeans, however, significantly facilitate the manner in which these issues can be approached - both from a design and imple-mentation standpoint. In combination, they have the potential to raise component model development, interoperability, and reusability, to a new level.

5 SUMMARY

The Java language extensions that constitute Silk were designed to encourage better discrete-event simulation through better programming by better programmers. Since the modeling language is integrated into the Java programming language, the full power and flexibility of the Java programming language is available. Unlike proprietary modeling environments, users also benefit from the growing number of commercially available professional Java development tools. Silk and JavaBeans also greatly advance both the state of the art and practice of visual modeling with reusable industry-specific and company-specific modeling components. These language-level and component-level advances in combination with the ability to distribute and execute models via the Internet will also foster increased activity in the development of high-level, domain-specific simulation tools that end-users favor.

REFERENCES

Burke, E. and R. Kilgore. 2000. Modeling web servers using Java and Silk. *Proceedings of the 2000 Summer Computer Simulation Conference*. SCS International, Ghent, Belgium.

Healy, K. and R. Kilgore. 1997. Silk^{TM}: A Java-based process simulation language. *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. Healy, D. Withers, and B.L. Nelson. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. A., Healy, K. J. and Kleindorfer, G. B. 1998. The future of Java-based simulation. *Proceedings of the 1998 Winter Simulation Conference Proceedings*, ed. D. J. Medeiros, E. F. Watson, J. S. Carson, M. S. Manivannan. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. and K. Healy. 1998. Java, enterprise simulation and the Silk^{TM} simulation language. *Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation*, ed. P. Fishwick, D. Hill, and R. Smith. SCS, San Diego CA..

Kilgore, R., K. Healy, and G. Kleindorfer . 1998. Silk^{TM}: usable and reusable Java-based object-oriented simulation. *Proceedings of the 12th European Simulation Multiconference*. SCS International, Ghent, Belgium.

Pidd, Michael , Noelia Oses and Roger J. Brooks. 1999. Component-based simulation on the web? In *Proceedings of the 1999 Winter Simulation Conference,* ed., P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Sawhney, Anil, Jayachandran Manickam and André Mund. 1999. Java-based simulation of construction processes using Silk. 1999. In *Proceedings of the 1999 Winter Simulation Conference,* ed., P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

AUTHOR BIOGRAPHY

RICHARD A. KILGORE is the President of ThreadTec, Inc., the developers and distributors of Silk. He has over 20 years of experience as a modeling consultant to Fortune 500 firms in a variety of industries with a variety of simulation and scheduling tools. He received his B.B.A. and M.B.A degrees from Ohio University and Ph.D. in Management Science from the Pennsylvania State University. Formerly, he was a capacity-planning analyst with Ford Motor Co. and Vice-President of Products for Systems Modeling Corp. His e-mail address is `<kilgore@threadtec.com>`.