Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

OBJECT-ORIENTED SIMULATION WITH SML AND SILK IN .NET AND JAVA

Richard A. Kilgore

SML Simulation Services
P. O. Box 7
Chesterfield, MO 63006, U.S.A.

ABSTRACT

This tutorial is for advanced simulation developers engaged in the use of object-oriented programming languages and libraries that support object-oriented, discrete-event simulation. The tutorial is based on generic structures common to SML simulation libraries in the .Net languages VB.Net, C# and J# and the Silk libraries in Java. The focus of the tutorial is on the use of consistent design patterns that encourage usability, reusability and cross-language compatibility. Particular emphasis is placed on designing and coding object-oriented simulation models to properly transfer simulation control between entities, resources and system controllers, and on techniques for obtaining a one-to-one correspondence between simulation code and system behavior.

1 INTRODUCTION

The tools for developing discrete-event simulation applications have evolved from libraries of simulation subroutines to high level process languages built on those libraries to visual development environments built on the process languages. Throughout this evolution, industrial-strength simulation applications continue to require the ability to add software functionality beyond that provided by off-the-shelf simulation packages. In particular, the ability to drop down into the underlying programming language is often necessary to achieve the desired system behavior and control policies.

The emergence of mainstream object-oriented programming languages like Java and C# and simulation developers trained in these languages has led to the development of a wide variety of object-oriented simulation libraries. Some libraries are personal toolkits of researchers, professors, graduate students and consultants. Other libraries are products of internal efforts by company developers. And still others are released as commercial libraries used for development of models or new simulation development environments. Since many of the SML libraries share common objectives, it is also natural that some common reusability and design patterns and interoperability objectives emerge as shown in Figure 1.

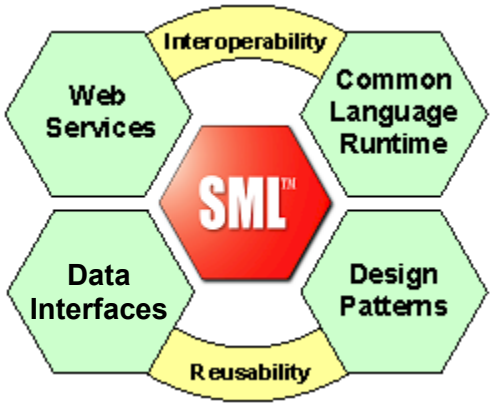

Figure 1: SML Interoperability and Reusability

This tutorial is an opportunity for advanced simulation developers using object-oriented languages to review the current status of the commercial simulation libraries for discrete-event simulation like SML (Kilgore 2002a, 2002b) and Silk (Kilgore et al. 1998a, 1998b). These object-oriented simulation tools presented are not compiled simulation languages but are rather extensions of the base programming language. The power, flexibility and extensibility of the languages derive directly from their base language and the ability to separate the data input and output interfaces from the modeling programming code in these languages (Kilgore 2001).

The remainder of this paper is intended to serve primarily as an introduction for those not yet familiar with the language level features of SML and 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 SML and the Silk language are listed in the References section and readers are encouraged to reference those web sites for more detailed specifications and examples. Section 2 contains an overview of creating object-oriented models with SML and Silk. Section 3 describes the development of SML and Silk in Java and .Net Integrated Development Environments. Section 4 contains concluding remarks.

2 OBJECT-ORIENTED DESIGN

Object-oriented simulation 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 entity class. Experience with this design continues to create better opportunities for reusability through production of the simulation "code" that is more readable by other developers and engineers participating in the development.

To illustrate this concept, consider the standard 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 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 simulation class for each intelligent system component. The representation of the Customer and Teller classes are described in Figures 2 and 3. The numbers in the text refer to the line numbers in these figures and color is used in electronic versions of this paper to distinguish keywords (blue), keywords (red), comments (green), and user-defined identifiers (black).

2.1 The Customer Class

The package of object-oriented classes referenced are identified in an "import" or "using" statements [1] and the user-defined class name Customer is declared as an extension of the Entity class [2]. The class structure consists of the data declarations [3-11] which will define the characteristics of the simulation entities created from this class and the default *process* method [12-26] that will change those entity characteristics as the state of the system changes. The essence of object-oriented simulation is the use of these Entity methods, statements, and other objects within this process method to represent exactly what behavior the real systems entity experiences.

Each instance of this Customer class is assigned two unique, user-defined attribute identifiers, *attArrivalTime*, *attServiceDelay* [4]. Since the simulation will be executed directly in the underlying language, these attributes can be

```
1.  using; or import;

2.  public class Customer extends Entity {

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

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

12. public void process( ){

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

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

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

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

25.    dispose( );

26.    }// end of process method
27. }// end of Customer class
```

Figure 2: Customer Class Definition

```
1.  using; or import;
2.  public class Teller extends Entity {

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

5.     public void process ( ) {

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

7.       // wait while condition is true (no customers in queue
8.       waituntil( condition (  Customer.queCustomer.getLength( ) > 0 ) );

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

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

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

18.   }// end of process method

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

Figure 3: Teller Class Definition

any valid language or user-defined type. The entity design is improved if this set of attributes mirrors the actual observable characteristics necessary for decisions in the real system, rather than simply a set of less descriptive programming flags.

While each Customer instance will have these unique attribute identifiers, all instances of the Customer class will share common *static* class variables of other language or simulation objects [6-11]. Only objects for random variable generation and statistics are shown in this example, but again remember that these models are programs so the modeler has great flexibility in the location and form of this information. For example, a more complex model might contain an array of all of the required processing delay distributions that this entity might require as a separate data object, thus removing the specifics of entity performance from the general representation of entity behavior.

A significant advantage of SML and Silk over previous object-oriented languages is the use of process-oriented methods familiar to users of other simulation language. Every 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 [12-26] 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 [14] the arrival of the next Customer using a sample from a Exponential random variable object created in the data declaration. The *attArrivalTime* variable is then set to the current value of simulation *time* [15]. The "att" prefix is not required and has no special 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 [17]. 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 if the basis for the service requirement is a characteristic of the customer, 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 models easier to reuse.

The Entity queue method then places this Customer instance in a *queue* [18] object which is simply an ordered

list of Customer entities. Note that this queue is not linked with any particular *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" through process methods 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 *halt* method [20]. This distinction may seem cumbersome at first and the traditional entity-push approach could be used throughout the process definition. But the object-oriented design requires that data characteristics and behavior of each object to be encapsulated within that object. The significance of this approach will become clearer as the behavior of the Teller object is described below.

The Customer is "pulled" from the queue and activated by the Teller object [shown in Figure 3, line 13]. The Customer object then continues the process method by recording the time spent in the halted state in a Observation statistic object [21]. The TimeDependent object for Customer queue length [11] 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 [23] until service is completed and the Customer is activated by the Teller object [Figure 3, 15]. Statistics for system time are then recorded for system time [24], and this instance of the Customer class is then disposed [25]. The dispose method actually places the entity object in a pool of Customer objects to be reincarnated as representations of future customers.

2.2 The Teller Class

The description of the Teller class is found in Figure 3. It defines the simulation system 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 an Entity [2]. A *Resource* object created to represent the Teller state [3]. The responsibility for when and how to change this state from busy to available is left to the *process* method for the Teller [5-18]. The *while* block [6] is used to continuously loop the single instance of the Teller throughout the simulation. By default, an entity executes the *process* method only once so this construct is necessary to allow the instance of Teller entity to continuously repeat the process method for subsequent Customers.

The *waituntil(condition( ))* construct in [8] combines the *waituntil* method and the *condition* method. Similar to the *halt* method, this statement temporarily stops the process of an 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 simulation or user-defined *state variables*.

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. Performance is less affected by this complexity as Boolean conditions are reevaluated only when those objects which appear in the methods change value. 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 [10]. 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. Users commonly "wrap" complex methods like these within other simpler user-defined methods of their own creation. But the power of open-source SML is the ability of the development community to create and extend the language without sacrificing the underlying power and flexibility of the basic Entity methods.

The Teller object uses the reference to the Customer entity *entCustomer,* to access the service delay attributes of the Customer [14] and to invoke the activate method to resume the process method for the halted Customer entity as described earlier [13,15]. The seize and release methods in [12,16] modify the busy state of the Teller Resource object to allow the TimeDependent object to automatically track Teller utilization [4].

2.3 The SIMULATION Class

The Simulation class shown in Figure 4 is necessary to schedule the arrival of the first instance of each class in the *init* method [3] which is called at the beginning of each simulation run. The *newEntity* method [5] is responsible for the creation and use of the entity 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 to start the execution of the desired number of runs and run length [12,13]. Execution will end with the creation of a Summary Report window or the user can ask

```
1.  using; or import;

2.  public class Simulation {

3.    public void init ( ) {

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

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

9.    } // end init method

10.   public void run ( ) {

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

15.   } // end run method

16. } // end Simulation class
```

Figure 4: Simulation Class Definition

that a *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

As seen in this example, object-oriented simulation involves flexibility regarding the choice of design patterns. Consider the decision earlier to declare the *Queue* object to be a characteristic of the Customer class [Fig. 1, 10]. Even in this simple example, a 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 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. 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 queuing behavior without modifying the Teller class.

3 DEVELOPMENT ENVIRONMENTS

The Silk and SML simulation extensions to object-oriented languages are themselves implemented entirely in the underlying languages. The only requirements for building and executing simulation models are a Java or .Net language compiler and Java or .Net run-time that are compatible with specification of the underlying language. Most commercial simulation software constrain users to a single proprietary and often cumbersome development environment. Commercial programming IDE's like Visual Studio .Net provide a sophisticated graphical interface and a rich collection of tools for project management, source code creation, modification, compilation, debugging, and deployment. Figure 5 contains a screen snapshot of the example problem from the previous section within an integrated development environment.

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 SML significantly facilitate the manner in which these issues can be approached - both from a design and implementation standpoint. In combination, they have the potential to raise component model development, interoperability, and reusability, to a new level.

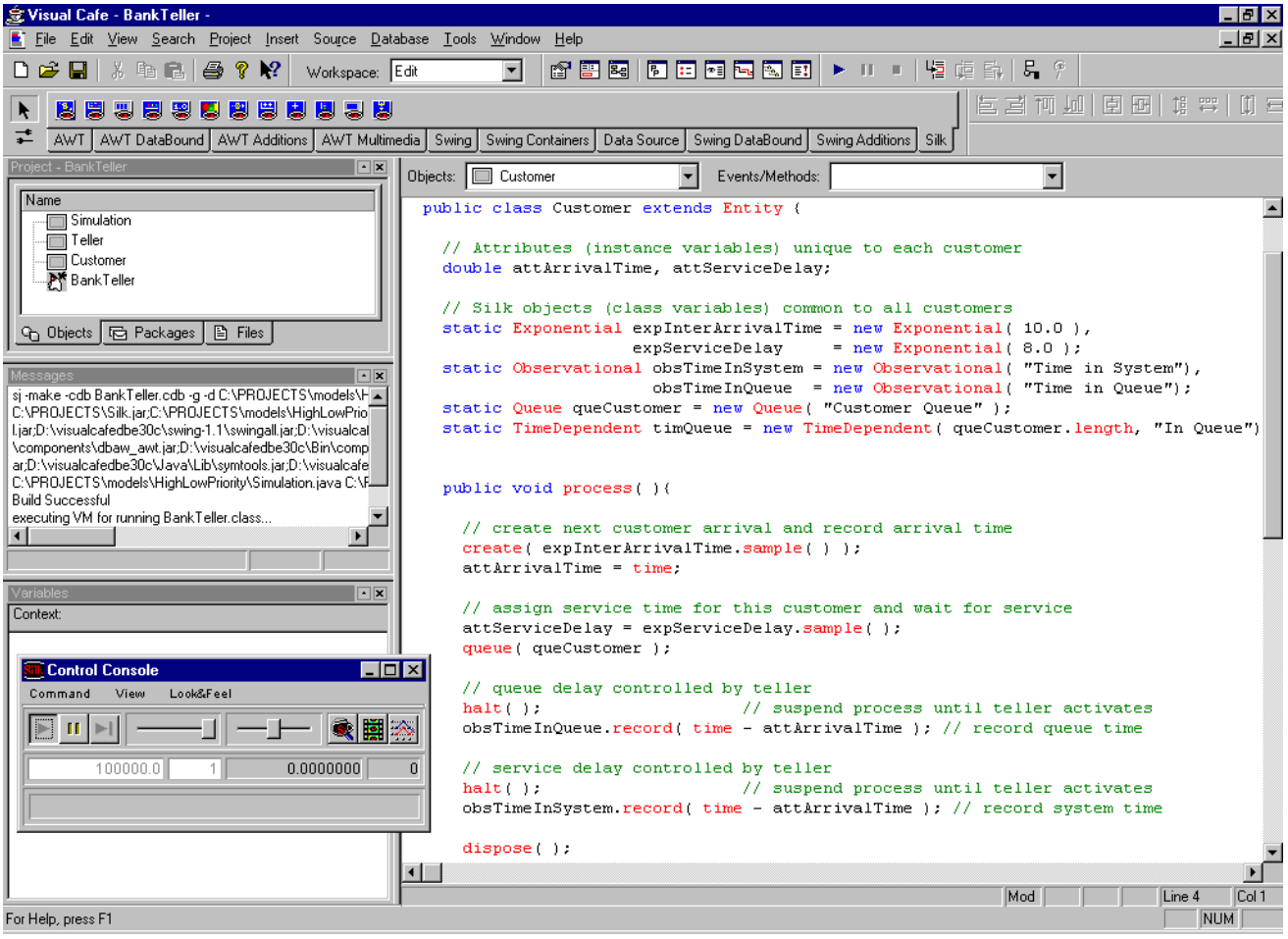

Figure 5: Modeling using an Integrated Java Development Environment

4 SUMMARY

The language extensions that constitute SML and Silk were designed to encourage better discrete-event simulation through better programming by better programmers. Since the modeling language is integrated into the programming language, the full power and flexibility of the programming language is available. Unlike proprietary modeling environments, Silk and SML users also benefit from the growing number of commercially available professional development tools. And unlike proprietary software, SML users can benefit from the large community of simulation researchers and practitioners who can guide and participate in SML development. The open-source licensing of SML will encourage developers to share language-level and component-level advances via the Internet and will also foster increased activity in the development of high-level, domain-specific simulation tools that end-users favor.

REFERENCES

Kilgore, R. 2001. Open-Source Simulation Modeling Language (SML). In *Proceedings of the 2001 Winter Simulation Conference,* ed., B. Peters, J. Smith. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. A. 2002a. Simulation Web Services with .Net Technologies. In *Proceedings of the 2002 Winter Simulation Conference,* ed., E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. A. 2002b. Multi-Language, Open-Source Modeling using the Microsoft .Net Architecture. In *Proceedings of the 2002 Winter Simulation Conference,* ed., E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. A., K. J. Healy, and G. B. Kleindorfer, 1998a. 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, 1707-1712. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

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

AUTHOR BIOGRAPHY

RICHARD A. KILGORE is a consultant in the development of industrial simulation and scheduling solutions. Dr. Kilgore 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., Vice-President of Products for Systems Modeling Corp and President of ThreadTec, Inc. His e-mail address is <mailto:kilgore@threadtec.com>.