Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

FUNDAMENTALS OF OBJECT-ORIENTED SIMULATION

Jeffrey A. Joines
Stephen D. Roberts

Department of Industrial Engineering
Campus Box 7906
North Carolina State University
Raleigh, NC 27695-7906, U.S.A.

ABSTRACT

An object-oriented simulation (OOS) consists of a set of objects that interact with each other over time. This paper provides an introduction to the fundamental OOS design elements by contrasting OOS with its procedural counterpart. It further addresses the important issue of composition versus inheritance that distinguishes object-based from object-oriented languages.

1 THE SIMULATION SOFTWARE CHALLENGE

There has been tremendous growth in the capability of computing hardware during the past three decades. The cost of memory, in-core and auxiliary (hard disks, flash cards, CDROM, etc.) has dropped dramatically while processor speed and capability has grown enormously. What was available in environmentally controlled rooms containing massive machinery in the 1960s was less powerful than most people now have at their fingertips. In the popular computer folklore is Moore's Law— the observation that the logic density of silicon integrated circuits has closely followed the curve (bits per square inch) = $2^{(t\ -\ 1962)}$ where t is time in years. This means the amount of information storable on a given amount of silicon has roughly doubled every year since the technology was invented. This relation, first uttered in 1964 held until the late 1970s, at which point the doubling period slowed to eighteen months. It has remained at that value through recent years.

For modelers, an important question is how can simulation can take full advantage of the computing power now available. Software engineering provides part of the answer. However, writing software "from scratch" is no longer advisable since software systems tend to be complex and several libraries exist for many of the common functions. Thus, simulation models need to include more than computational efficiency if they are to have wider utility and acceptance in a multi-media, virtual reality, and graphical interface software world.

Modeling and software have had a symbiotic relationship. The computer is more than a computational engine for simulation algorithms and should be regarded as a tool for modeling. The technology of simulation is now a mature and developed methodology. Although there is plenty of room for additional research on fundamental areas (e.g., random number and variate generation, the next event simulation process, reliable and appropriate statistics), there are now widespread adoptions and use of computer simulation techniques.

However, the real limits on the future adoption of simulation may rest on our ability to represent complex systems and to do it easily, which can be construed as a matter of modeling style. The purpose of this paper is to describe ways to improve modeling style – through object-oriented simulation and to describe the fundamentals of object-oriented simulation. It is useful, however, to first consider the matter of programming style – modeling style usually follows programming style.

2 PROCEDURAL STYLE

As long as you can specify statement sequences, define variables, do branching, perform iteration, and have I/O you can do everything a Turing machine can do which in turn means everything a computer can do. Thus the distinction in programming style is not what can and can't be done but what can be done easily. However, what can be done easily may be a matter of judgement.

Early programs were long sequences of labeled statements (simulation instructions) where movement of program control used labels. Ironically, many fairly recent simulation languages maintain this same approach. Look, for example, at one of the source files from GPSS:

```
    GENERATE      1,,,,1
    ASSIGN        1,V$DMND
    TEST GE       X$STOCK,P1,TRUBL
    SAVEVALUE     STOCK-,P1
TAB TABULATE      STOCK
    TABULATE      LOSES
```

Owing to the repetition of some logical procedures, functions or subprograms were added to programming languages. To give these functions generality, argument lists were added that could change the computation within a function from call-to-call. Using functions to subdivide a programming problem gave rise to the notion of "functional decomposition," which remains today a popular approach to programming and simulation modeling. For example, in programming a simulation, a random variable generation library of functions could provide a means to obtain a sample from a random variate generator as:

```
double  normal(mean, standDev, randomNumber)
```

GASP was almost totally composed of functional libraries that were called by user written code, as illustrated by the following Fortran based GASP IV code published in 1974:

```
103 IF (NEXT) 107, 108, 104
104 CALL COPY (NEXT)
    CALL FILEM (NFRA)
    ICS=NEXT+NNAPO
    NEXT=NSET(ICS)
    GO TO 103
```

GPSS, SLAM, and SIMAN were the simulation versions of the "library" approach to simulation, but the libraries were invoked much differently and invisibly to the user. Instead of writing general purpose programming code, users constructed text files containing a sequence of simulation "instructions." This approach provided a higher level of abstraction than programming in a low-level language, making it easier to model complex systems. An example from SLAM II published in 1995 is:

```
    CREATE, EXPON(30),,1,,2;
    ACT,,,AS1;
    ACT,,,AS2;
AS1 ASSIGN,ATRIB(2)=1,ATRIB(3)=DPROBN(2,1),
    ATRIB(4)=8,ATRIB(5)=60;
    UNBATCH,3;
Q1  QUE(1),,,,ASM1;
```

These instructions generally have a direct correspondence to a form of a flowchart (also called network). While such input makes it easy to specify the simulation, it limits the direct impact the modeler can have on the execution of the simulation, since these simulation instructions did not constitute a programming language. Instead these are generic model templates for simulation.

Thus, many models written in earlier versions of these languages were highly augmented by general programming code (Fortran, C, etc.) containing function calls to the simulation libraries, since function calls could not be directly invoked and users could not write functions, with the native simulation instructions. In particular, Visual SLAM continues to promote extensive use of programming "inserts" with Visual Basic and C.

It is important to note that while simulation "languages" like GPSS, SLAM, and SIMAN were easy to use, although limited, there were more powerful simulation programming alternatives. For example SIM-SCRIPT and SIMULA were full programming languages with simulation functionality built into the language grammar and syntax. Using these languages, users "programmed" the simulation. SIMULA was not widely appreciated at the time as a simulation language but would, in fact, form the basis and motive for much of the modern object-oriented paradigm.

In all cases (except for SIMULA), the style was procedural based. A problem was decomposed into procedures and either represented by general components, like a queue, or represented in programming code with data structures and code. Procedural programming represents today a fundamental style of programming usually learn-ed in the first exposure to programming or modeling.

There are several fundamental problems with using the procedural style of modeling and simulation. Procedures do not correspond to real world components. Instead, they correspond to methods and algorithms. Therefore acts engaged by entities must be given a context for procedures to be easily specified. Many simulation contexts are based on networks of queues (often complicated queuing situations). The modeling approach is to let the queuing network create the procedural structure that is traversed by entities. When this structure is appropriate, as it often is in a manufacturing or communications application, the model is a convenient analog to the real system.

However modeling languages are limited when confronted with complicated circumstances, such as the need to code an algorithm that creates a schedule based on anticipated volume and current use of facilities. It is then that the need for general programming manifests itself. However there is a fundamental difficulty in communication between the simulation code, provided by a simulation vendor, and user code from a general programming language. The only means of communication is generally through global data exchange or function calls. These mechanisms are vulnerable to inappropriate use and were dangerously visible to users.

Perhaps the greatest limitation of the procedural style is its lack of extensibility. From the earliest simulation languages until the early 1990s, the only way to adapt these simulations was through functional extension. In other words, you could add structural functionality to the simulation but not alter any of its basic processes, like giving properties to resources. For example, if you needed the simulation to include a bridge crane, you had to program it completely yourself or model it with the features available. One of the reasons for this lack of extensibility was that procedural changes were the only approach to model changes. Specifically, vendors had no way to partially hide implementation details and were either forced to give access to source code or restrict the access to the features. A module or file provided a form of encapsulation (which more recent simulation languages call templates or subnetworks), but these collections do not provide for autonomous objects.

3 OBJECT STYLE

The class concept evolved out of the notion of encapsulation, an idea that originated in SIMULA. However SIMULA viewed objects as much more than encapsulation. Objects needed independence of action and a means to hide their implementation details, yet provide an interface for their use. Further, there needed to be way to construct objects and to communicate among them. C++ borrowed all these ideas from SIMULA (as did Smalltalk) and put them into the procedural programming language C. We use C++ to illustrate the object style.

3.1 An Example: The Exponential Random Variable

Suppose you are modeling an exponential random variable in a simulation. The random variable may be described by a standard exponential statistical distribution, which has a set of parameters (e.g., a mean in this case). This mean would be considered an attribute of the exponential random variable object. It may be important to obtain observations from this random variable via sampling. One may want to obtain antithetic samples or to set the random seed. Sampling from the exponential random variable defines a particular behavior**.**

3.2 Encapsulation

The entity "encapsulates" the properties of the object because all of its properties are set within the definition of the object. In our example, the exponential random variable's properties are contained within the definition of the random variable so that any needs to understand or revise these properties are located in a single "place." Any users of this object need not be concerned with the internal "makeup" of the object. This also facilitates the ability to easily create multiple instances of the same object since each object contains all of its properties.

In C++, the keyword "class" is used to begin the definition of an object followed by the name of the object class and then the properties of the entity are defined within enclosing {}. For example, the following object defines the `Exponential` class.

```
class Exponential{
…// Properties of the Exponential
};
```

Without encapsulation, properties could be spread all over, making changes to the object very difficult.

3.2.1 Class Properties

The class definition specifies the object's properties, the attributes and behaviors. The attributes define all the singular properties of the object while the behaviors define how the object interacts within the environment with other objects. Attributes are considered the data members of an object. In the case of our `Exponential` random variable, its mean (given by the identifier `mu`) would be a real number attribute.

```
double    mu;
```

Other attributes would be similarly defined.

The behaviors (sometimes referred to as methods) of an object represent actions the object can perform or take. For example, if the exponential random variable needed to obtain a sample, the following member function can be used:

```
double   sample(){
   return –mu * log( 1.0 – randomNumber() );}
```

where the `randomNumber()` function yields a uniform random variable between 0 and 1. By representing behavior with functions, the object can react to parameters passed in the function argument as well as change variable values within the function.

3.2.2 Classes and Instances

Notice, the word "class" not "object" is used in defining the object, which can be confusing, since it would seem that we are defining objects. Lets consider the more complete definition based on our prior discussion of encapsulation and properties (ignore the "public" for now), the `Exponential` class is defined as follows.

```
class Exponential{
   public:
      double mu;
      double sample(){ return –mu * log( 1.0 –
                         randomNumber() );}
};
```

Rather than defining an object directly, a class is defined where the class provides a "pattern" for creating objects and defines the "type." By defining a class (of objects), rather than a single object, the opportunity exists to use the class to create many objects (i.e., re-use existing code). Furthermore, as seen later, the class is a description of a pattern for constructing objects which can be easily extended. Now, objects can be created directly from this class once defined. These created objects are called "instances" of a class. For example, `serviceTime` is an instance of the `Exponential` class.

```
Exponential   interarrivalTime, serviceTime;
```

3.3 How Do Objects Communicate?

An OOS models the behavior of interacting objects over time. However before we can consider a simulation, we need to understand how objects interact or communicate with each other. The interaction among objects is performed by communication called "message passing." One object sends a message to another and the receiving object then responds. An object may simply publish a message that may be responded to by one of several objects. For example in a bank simulation, a customer arrives at a bank and may be served by any of several tellers. In a O-O context, the customer publishes their arrival and waits for service by a teller. There are several ways in transmitting messages in an object-oriented program and it depends on the programming language.

3.3.1 Direct Reference

Perhaps the simplest form of message passing is direct reference to the object's attributes or data members. For example, if the `interarrivalTime` object needed to have a mean of 5.5, then the simplest means to communicate this message is through direct assignment.

```
interarrivalTime.mu = 5.5;
```

This message causes the object to receive the value and set its variable `mu`. This is a forced message because the object has no choice but to perform the action.

3.3.2 Data Methods or Functions

Rather than forcing a value upon an object, a value could be communicated to the object and then let it determine how to deal with the value. For example, if a new "member function" or data method to the Exponential class called `setMu()` was added as follows.

```
void setMu( double initMu ){ mu = initMu; }
```

Now the object is sent the `setMu` message with a message value of 5.5 which "communicates" our interest in changing the mean and `interarrivalTime` receives the message and changes its internal value of `mu`.

```
interarrivalTime.setMu(5.5);
```

Although this example really does the same thing as the direct reference, there are important distinctions. First, in our function call we simply "passed" the value of 5.5 to the object. Second, we didn't tell the object how to change the attribute `mu`. The object has a function written by the designer of the `Exponential` class that causes the mean parameter to change. Notice the user of the function does not need to know how the function inside the class works. In fact, the class designer could change the internal name of `mu` to `expMean` within the class, and all exiting user code would remain the same. This encapsulation of the data is extremely important in OOS. Also, the same message can be made to respond to several different message value types is often referred to as "polymorphism."

3.3.3 Pointers

Another way to communicate is indirectly through pointers that are simply addresses of the location of an object. For example, a pointer to the `interarrivalTime` object can be created as follows and the setMu message can be sent via the pointer.

```
Exponential * rnPtr = &interarrivalTime;
rnPtr->setMu(5.5);
```

Pointers have the advantage of not needing to know the particular object ahead of time, but only the address of the object. Thus, if we change the pointer to point to the `serviceTime` object, the format of the message remains the same. With a more complex message, use of pointers becomes very convenient.

```
RNPtr->setMu(3.5);
```

3.4 How Are Objects Formed?

In our example, the exponential object has no ability to be created with different means. Instead, the object's mean was changed to a specific value. Although an object can be instantiated from a class without special instructions, often we want the creation to accomplish certain objectives. Likewise, we also might want to do something special when an object is destroyed.

3.4.1 Constructors and Destructors

Special member functions can be defined that act when an object is created and destroyed which are called constructors and destructors, respectively. The constructor is recognized by having no return type and the same name as the class. For example, the following could be a constructor for the exponential object.

```
Exponential( double initMu ){ mu = initMu; }
```

This function accepts the invocation argument and sets the internal mean to it. An object whose initial mean is 4.3 can be specified upon creation as follows.

```
    Exponential   serviceTime(4.3);
```

In C++, functions can be "overloaded" so that they differ only in their formal arguments (i.e., "polymorphism"). Therefore, a class can have multiple constructors. For example, if we wanted the exponential to accept an integer specification of its mean.

```
    Exponential( int startMu ){ mu = startMu;}
```

Now, exponential objects with either a double or an int as arguments can be specified (actually C++ will make appropriate conversions among its built-in types but this example illustrates the way a user could provide conversions among user-defined classes). The following creates two objects using different argument types.

```
  Exponential   arrival(9.3), inspect(6);
```

Users can also define a special member function called a destructor that acts when the object is destroyed. Only one destructor can be defined since it has no arguments. For example, a destructor for the exponential class has the following form.

```
  ~Exponential(){// print out how often used? }
```

3.4.2 Visibility of Properties

It should be clear that a user of a class does not really need to know the internal workings of the class. For example, they do not need to know what algorithm is used to obtain the sample (they may want to know for their own assurance). Furthermore, the designer of the class may not want the user of the class to know everything about the class. Thus, the class designer has the option of causing properties of the class to become invisible to users of the class and to provide a public interface to those hidden properties. The two most frequently used labels are "public" and "private." Properties within a class that are public can be accessed directly by a user while those that are private are available only to the designer. For example, the variable containing the mean is made private within the class to prevent improper use (i.e., direct manipulation). Our class would then look like the following.

```
class Exponential{
 private:
  double mu;
 public:
  Exponential(double initialMu ){mu=initialMu;}
  double sample();
  void setMu( double changeMu ){mu = changeMu;}
  double getMu( ) { return mu; }
};
```

Now `mu` cannot be changed directly by a user. Thus the direct reference to `mu`, as done earlier, will fail. Communication to the exponential objects must be performed through member functions. The designer of the class can now protect the class data members from unwanted changes while the user of the class is unaffected.

3.5 How Are Objects Formed From Others?

One of the fundamental benefits of an O-O design is the ability to make other objects out of existing ones. We have already seen how to design a class of objects using the built-in types from C++. Suppose the following random number class has been defined which generates uniformly distributed numbers between 0 and 1.

```
    class RandomNumber{
          long   seed;
       public
          RandomNumber( long seed = -1);
          void setSeed(long sd){seed=sd;}
          virtual double sample();
    };
```

In this definition, the constructor argument can be specified or left blank to default to their initial values (i.e., -1 means use the next seed). The public member function `sample`() is used to obtain a sample and we will assume that the seed will be updated appropriately with each call. The "virtual" keyword will be discussed later.

There are two ways this random number generator could be used with our `Exponential` class. The first method is called **composition**, in which a random number object is included within the exponential class. The second method of using the random number generator is through **inheritance** which makes the exponential class a kind of random number. Inheritance is one of the major features that distinguish a "object-based" language from a true "object-oriented" one.

3.5.1 Composition

First, consider the case of composition where we simply compose the new class from the existing class:

```
    class Exponential{
       private:
          double mu;
          RandomNumber rn;
       public:
          void setSeed(long sd){rn.setSeed(sd);}
    … //Public Properties
    };
```

Notice that the `Exponential` is defined simply to "have" a `RandomNumber`. In O-O parlance, the relationship between the `Exponential` and the `RandomNumber` rn is called a "has-a" relationship. The data member rn is used in the `sample`() function of the exponential. Notice, a `setSeed`() needs to be defined in order to access the one in the random variable.

3.5.2 Inheritance

The second kind of relationship among classes is called an "is-a" relationship and is based on inheritance or a parent-child relationship. In our example, the exponential can be considered a kind of random variable. It would be useful for the `Exponential` to be a child of `RandomNumber` and thus inherit all the random variable properties. Hence, what could be done to the random variable could also be done with the exponential. No additional `setSeed`() is required since the one in the random class can be used.

For example, sometimes a sample from an exponential is needed while other times a basic uniform generator is required. Suppose the following two objects and pointer are defined:

```
RandomNumber   uni;
Exponential   exp(5.5);
RandomNumber * pRN = &uni;
```

If at an activity in our simulation, a sample from a random variable is needed, the following message is sent to obtain an activity time.

```
pRN -> sample();
```

However, because Exponential is also a `RandomNumber`, the pointer `pRN` could be assigned to either an `Exponential` or a `RandomNumber` and the same message applies. In the **composition** example, two separate activities would be required (i.e., one which used an exponential and another one which used a uniform).

```
pRN = &exp;
```

In a true O-O language with inheritance, the message would be sent to the proper object and the sampling would be from the correct sampling function. In O-O terms, determining which variate to sample at run-time is called "run-time" binding and is performed by specifying the `sample`() to be "virtual" in the parent class.

To specify that `Exponential` inherits from `RandomNumber`, the header for the class definition would be modified as:

```
class Exponential: public RandomNumber{...
```

showing that `RandomNumber` is the parent and its visibility is "public".

Under inheritance, the child class inherits the public (and protected) properties of the parent. Now these properties are directly available to the child class and the class type resolves any conflicts. C++ also permits multiple inheritance, meaning a child can inherit from several parents.

4 OBJECT-ORIENTED VS. OBJECT-BASED

Because many simulation languages offer pre-specified functionality produced in another language, the user cannot access the internal function of the language. Instead, only the vendor can modify the internal functionality. Also, users have only limited opportunity to extend an existing language feature. Some simulation languages allow for certain programming-like expressions or statements, which are inherently limited. Most languages allow the insertion of procedural routines written in other general-purpose programming languages. None of this is fully satisfactory because, at best, any procedure written cannot use and change the behavior of a pre-existing object class. Also, any new object classes defined by a user in general programming language do not co-exist directly with vendor code.

4.1 Object-Based Extension

The object-based approach only allows extensibility in the form of composition (i.e., new objects can only be created out of existing objects). The simple `Event` object will demonstrate the limitations of extensibility only through composition. The Event object is used to move the simulation from one time to the next. Events are placed on the calendar and, when an event is removed from the calendar, the `processEvent`() function is called to handle the event. The following gives a portion of the `Event` class that can be used to process arrival of entities into the network and end of service events. Notice that depending on the type of event, the appropriate event handling function is called. This is an example use of composition.

```
class Event{
private:
   double eventTime, eventType;
   Source *source;
   Activity *activity; //… More properties
public:
   void processEvent(){
     select EventType{
      case ArrivalEvent:
        source->newArrival(Entity);  break;
      case EndofService:
        activity->endofService(Entity); break;}}
   //… Additional Properties
};
```

If the user wants to add additional events (e.g., a monitor event), it would require the designer to add an appropriate data member, data methods, and then provide an additional case statement. Therefore, the designer has the impossible problem in anticipating every kind of event.

4.2 Object-Oriented Extension

An object-oriented simulation deals directly with the limitation of extensibility by permitting full data abstraction. Data abstraction means that new data types with their own behavior can be added arbitrarily to the programming language. When a new data type is added, it can assume just as important a role as any implicit data types and can extend existing types. For example, a new user-defined robot class can be added to a language that contains standard resources without compromising any aspect of the existing simulation language, and the robot may be used as a more complex resource. There are two basic mechanisms in C++ that allow OOS to provide for extensibility: **inheritance** and **genericity**.

4.2.1 Inheritance

Inheritance allows classes to exploit similarity through specialization of parent classes (i.e., child classes inherit the properties of the parent and extend them). All event types have an associated `eventTime` and `eventType` and the appropriate data methods to specify these properties. Therefore, specific event types would inherit these properties and provide additional ones (see Figure 1).

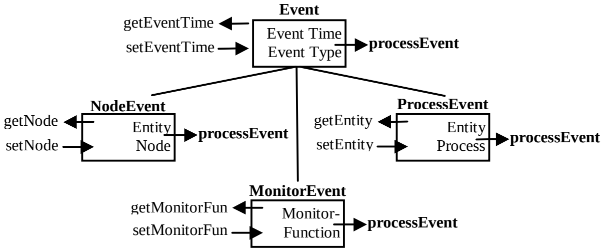

Figure 1: Inheritance Hierarchy

For example, `NodeEvent,` which provides events that occur at nodes (e.g., end of service at an activity), provides a pointer to the `Node` of interest and the `Entity` which caused the event. The `processEvent()` is declared virtual so that the appropriate `processEvent` is fired when the event is pulled off the calendar. The Event's `processEvent`() is a pure virtual function meaning any child classes must re-define it. The `NodeEvent`'s invokes the nodes `executeLeaving`() (another virtual function in the node hierarchy).

```
//Event’s processEvent
void virtual processEvent() = 0

// ProcessEvent’s processEvent
void virtual processEvent(){
  processPtr->executeProcess(entityPtr);}

//NodeEvent’s processEvent
void virtual processEvent(){
      nodePtr->executeLeaving(entityPtr);}
//ExecuteLeaving -virtual function in Node
```

Now the designer does not have to anticipate every type of event. Users have the ability to define their own events provided they inherit from an existing event class and provide an appropriate `processEvent`() function.

Unlike Java, C++ provides for multiple inheritance that facilitates a very useful and powerful feature with some subtle idiosyncrasies. Multiple inheritance allows you to combine the collection of data and behavior of several classes. For example, when modeling a textile distribution network, there are nodes that are vendors, distribution centers (DCs), and stores. Vendors are suppliers that ship garments to consumers while stores are strict consumers that receive shipments. However, DCs are considered both suppliers and consumers (i.e., DCs can supply other DCs and stores while receive shipments from other suppliers (either DCs or vendors)). In a single inheritance hierarchy, the designer must repeat similar code for either the supplier or consumer behavior or force an unnatural inheritance hierarchy.

4.2.2 Parameterized Types

Even with inheritance, many O-O languages like Java and Smalltalk can still be limiting in terms of extensibility. Eiffel and C++ provide an additional method of extensibility called genericity or parameterized types (i.e. templates). Parameterized types are special forms of composition that exploit commonality of function. For example, most simulations would declare a source object that is used to place entities into the network. In an OOS environment, the user may want TVs or Orders to arrive rather than generic entities. The user can create several different source nodes by inheriting from the base Source class as seen in Figure 2. Each of the new classes defines a new type of object to be created (i.e., TV, Order) and the "virtual function" `executeLeaving`.

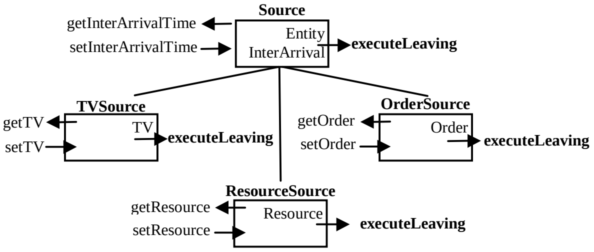

Figure 2: Inheritance Hierarchy versus Commonality

Notice, only the `Interarrival` object and methods are re-used in the child class. Each child class must define its own `executeLeaving`() when the only difference is the type of object released into the network. When objects provide the same functionality, parameterized types are used (see Figure 3). Now, the user specifies the type of entity to be released into the network and all remaining code is used. This is further demonstrated when a user wants to add statistics to the source node. The user only has to inherit from one class rather than create TVSourceStat, OrderSourceStat, etc.

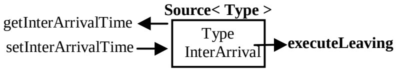

Figure 3: Parameterized Type

The following would declare two different source nodes.

```
Source<TV> tvSource(…);
Source<Order> orderSource(…);
```

5 CREATING A SPECIFIC OOS

A key to the creation of a fully integrated simulation package is the use of a *class inheritance hierarchy*. The formation of such a hierarchy is described in Joines and Roberts (1996). Object-based “frames” are used to collect classes into levels of abstraction. A *frame* is a set of classes that provide a level of abstraction in the simulation and modeling platform. A frame is a convenient means for describing various “levels” within the simulation class hierarchy and is a conceptual term.

While frames provide a convenient means to describe the levels of abstraction within the entire object-oriented simulation platform, another means of encapsulation is to place higher level complex interactions into “frameworks.” For our purposes, *frameworks* are used to describe those collections of classes that provide a set of specific modeling facilities. The frameworks may consist of one or more class hierarchies. These collections make the use and reuse of simulation modeling features more intuitive and provide for greater extensibility.

Special simulation languages and packages may be created from these object classes. For more information, see Joines and Roberts (1997) in the creation of YANSL, which is just one *instance* of the kind of simulation capability that can be developed within an OOS environment.

6 FINAL THOUGHTS

Modeling and simulation in an O-O language possesses many advantages. As shown, internal functionality of a language now becomes available to a user (at the discretion of the class designer). Such access means that existing behavior can be altered and new objects with new behavior introduced. The O-O approach provides a consistent means of handling these problems.

O-O systems view the world as a set of autonomous agents that interact or work together to solve some complex task. Each object is responsible for a specific task that helps one organize the complexity of complex systems which simplifies the computer programming tasks. O-O designs yield smaller systems through the reuse of common mechanisms. They are more reliant to change and are better able to adapt over time. O-O designs greatly reduces the risk of building complex software systems because they are developed to evolve incrementally from smaller systems.

The O-O ideas have re-rooted in simulation, after being initiated by simulation through SIMULA. The Smalltalk environment is fully O-O and contains fully OOS. Obviously simulation languages based on C++, like C++/CSIM and C++SIM, possess all the object-oriented capability described in this paper. Simple++ and MODSIM III are further examples of object-oriented languages that employ most of these concepts within different simulation frameworks.

The queuing network based languages like Arena and AweSim have beginnings of object-based features. Both languages provide a composition approach to creating network macros, through Arena templates and AweSim subnetworks. However neither are autonomous and independent objects in the sense described here and extensibility cannot be used to extend the active entities. Both have access to Visual Basic, which is itself only object-based. AweSim wraps its functionality in a few objects, whereas Arena contains a complete object model that is integrated with Visual Basic.

A new simulation language called SLX from Wolverine Software provides a new object-based simulation product from the makers of GPSS/H. This language has all the object-based facilities but has none of the object-oriented facilities. It does contain an extended macro facility for adding statements and extended features for representing the simultaneous behavior of objects.

To take full advantage of object-oriented simulation requires more skill from the user. However, that same skill would be required of any powerful simulation modeling package, but with greater limitations.

REFERENCES

Joines, J.A. and S. D. Roberts. 1996. Design of object-oriented simulations in C++. In *Proceedings of the 1996 Winter Simulation Conference*, ed., John Charnes, Douglas Morrice, Dan Brunner, and James Swain, 65-72. Institute of Electrical and Electronics Engineers, San Diego, CA.

Joines, J.A. and S. D. Roberts. 1997. An Introduction to Object-Oriented Simulation in C++. In *Proceedings of the 1997 Winter Simulation Conference*, ed., Sigrun Andradottir, Kevin J. Healy, David H. Withers, Barry L. Nelson, 78-89. Institute of Electrical and Electronics Engineers, San Diego, CA.

AUTHOR BIOGRAPHIES

JEFFERY A. JOINES is a Research Associate in the Furniture Manufacturing and Management Center at NCSU. He received his B.S.I.E, B.S.E.E, M.S.I.E and Ph.D. from NCSU. He is a member of INFORMS, IIE, and IEEE. His interests include O-O simulation, cellular manufacturing, and genetic algorithms.

STEPHEN D. ROBERTS is Professor and Head of the Department of Industrial Engineering at NCSU. He received his B.S.I.E., M.S.I.E., and Ph.D. from Purdue University. He was the recipient of the 1994 Distinguished Service Award. He has served as Proceedings Editor and Program Chair for WSC.