AN ABSTRACT OF THE THESIS OF

Joseph A. Fisher for the degree of Master of Science in Industrial and Manufacturing Engineering presented on August 24, 1992.

Title: Object Oriented Simulation Tools for Discrete-Continuous, Stochastic-Deterministic Simulation Models

Abstract approved: _____

R. Bruce Taylor

In this thesis an introduction to simulation and object oriented programming discusses the need for the creation of several classes which directly support object oriented simulation. The author places no restrictions on the type of simulations that can be conducted and simulation practitioners will find that the classes provided lay the groundwork for a robust object oriented simulation language.

In order to appreciate the development of the simulation classes, three examples from the literature demonstrate their use in continuous deterministic and discrete stochastic simulations.

In addition to the creation of several basic simulation constructs, two advanced simulation tools were developed. Inter-object communication and intelligent simulation capabilities will increase the power and flexibility available to simulation modeler's. Programmers, scientists, and engineers will appreciate the object oriented simulation engine and generic simulation objects that were created.

Object Oriented Simulation Tools for Discrete-Continuous, Stochastic-Deterministic Simulation Models

by

Joseph A. Fisher

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 24, 1992
Commencement June, 1993

APPROVED:

Redacted for Privacy

Assistant Professor of Industrial and Manufacturing Engineering in charge of major

Redacted for Privacy

Head of department of Industrial and Manufacturing Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented August 24, 1992

Typed by Joseph A. Fisher

ACKNOWLEDGEMENT

This thesis could not have been completed without the help of several people. First, Daniel P. Barduzzi and Earle M. Chiles, gave me support and encouragement throughout my education. Robert R. Safford advised me early in my graduate studies. Marshall J. English and Sabah Randhawa introduced me to computer simulation and operations research. My major professor R. Bruce Taylor assisted me in brainstorming solutions to my object oriented simulation problems.

I would also like to thank my minor professor Logen Logendran, committee members Eugene F. Fichter and John P. Bolte, and graduate council representative Shih-Lien Lu, on their suggestions for my final library copy.

Finally, credit should be given to my co-worker Douglas H. Ernst, for his persistence to quality in scientific modeling, and John P. Bolte my committee member and major contributor to the successful completion of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX FIGURES

PREFACE

Computer source code may be used in any project without permission of the authors (Fisher & Bolte, 1992) if reference to the original source is documented. Electronic transmission of selected parts of the source code can be obtained via anonymous FTP from bikini.cis.ufl.edu (128.227.224.1) in file pub/simdigest/tools/fisherj.zip

The classes and source code provided in this thesis were developed by Fisher and Bolte. The author (Fisher) under the guidance of Bolte, developed all stochastic elements, report templates, random number generators, generic simulation classes, and inter-object communication capabilities. Fisher provides the three examples as verification to a accurately working simulation engine.

OBJECT ORIENTED SIMULATION TOOLS FOR DISCRETE-CONTINUOUS, STOCHASTIC-DETERMINISTIC SIMULATION MODELS

1. INTRODUCTION

The computer has been a tool for scientists and engineers for the last 40 years. Numerical methods were born with the FORTRAN programming language in the late 1950's. At this time, scientists and engineers had a new powerful tool that allowed them to model complex mathematical relationships. Both hardware and software tools continue to improve and evolve.

The first object oriented simulation language (OOSL), Simula, was introduced in 1967. Since that time many new OOSL's have been created. Simulation seems to be the most natural application for object oriented programming (OOP).

The inclusion of a significant amount of tutorial material on simulation and object oriented programming has been included in this thesis to provide the reader with an evolutionary perspective of past research, and what may be the next generation of mathematical modeling "object oriented simulation."

The three goals of this thesis were to; one, develop a simulation environment that provided a better or easier way to do intelligent simulations, two, create a form of inter-object communication without introducing object dependencies, and three, produce simulation class constructs that would contribute to the creation of a robust object oriented simulation language.

So what makes this work unique, special, or any different than the countless OOSL's produced by academics, industrialists, and energetic programmers around the world? This research establishes a procedure for including intelligent simulation objects into a simulation model. It also includes the creation of several useful simulation class constructs. All of this work should lead to the development of a robust object oriented simulation language.

The ability to include artificial intelligence (AI) techniques into a simulation language has never been done in a straight forward manner. The problem in the past has been that languages that were good for simulation tended not to be good for AI techniques and vice versa. Recently, new object oriented languages have been developed which make the inclusion of AI into simulation languages feasible.
The on going development of computer hardware and software is necessary if we are to continue to answer questions about complex systems. Good hardware and software are a prerequisite to building good simulation models. Industry continues to produce good hardware but software has traditionally lagged behind. This thesis attempts to fill a gap in the simulation community by producing simulation tools with advanced features that will make simulation more powerful and more flexible. Like hardware, simulation software must continue to improve and evolve.

It is not the intention of this thesis to produce a commercial grade simulation language but to layout the basic components of such a language. To define a reasonable amount of work that could be accomplished in a period of two years, the scope of this thesis was limited to the items that were considered necessary. Continuous, discrete, stochastic, and deterministic simulations were considered essential. With the different types of simulations specified, a core simulation engine was developed to handle these different types.

Figure 1 shows the template for the creation of this OOSL. The main components of the simulation language would be the, Simulation Engine, Simulation Model, and Simulation Experiments.

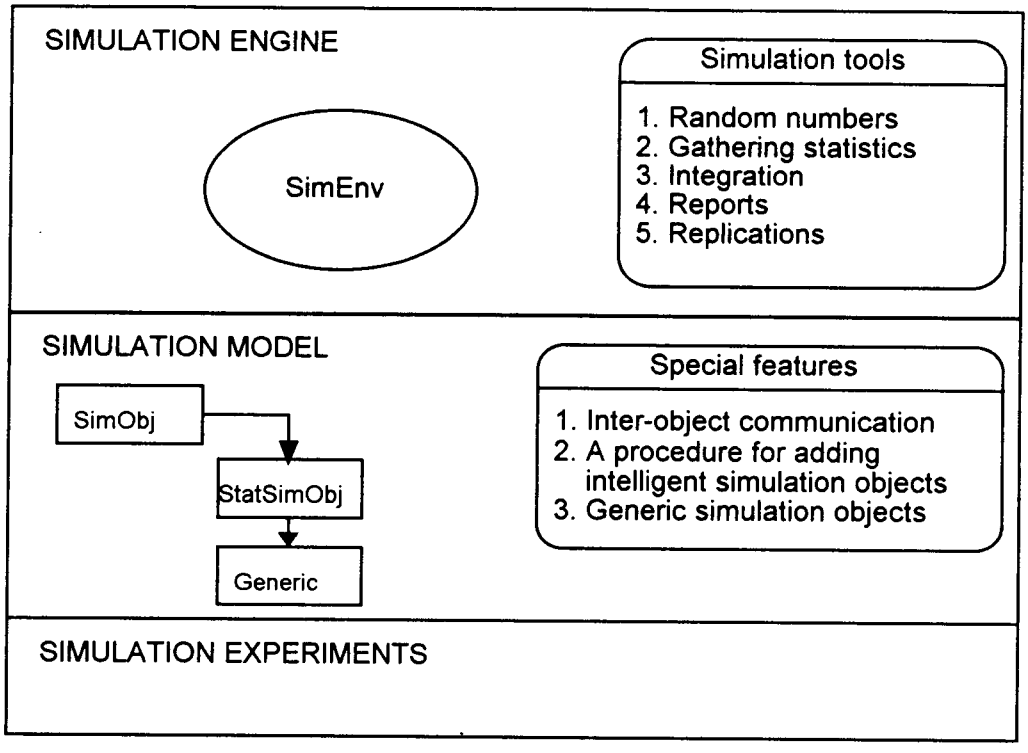


Figure 1. Template for the creation of an Object Oriented Simulation Language.

The Simulation Engine consists of the simulation environment (SimEnv) and several simulation classes which support a wide variety of simulations. Users of the Simulation Engine need to provide a Simulation Model for the Simulation Engine to simulate. Three examples and several generic simulation objects will help users design their own simulation objects.

Providing experimental design capabilities into a simulation language would be a very useful tool. Unfortunately, time limitations negated the development of an experimental design class. Perhaps others who use this engine will consider this challenge.

2. REVIEW OF LITERATURE

2.1 The history of simulation

During the years that followed W.W.II, a tremendous technological revolution took place. Companies provided a more diversified range of products and services. Manufacturing, in turn, became increasingly complex. Middle management used mathematical techniques perfected during W.W.II to solve problems of project planning, inventory control and capital investment. Operations research techniques experienced outstanding growth and interest during the industrial revolution.

With continuing opportunities for upper management to expand and modify old production processes, decisions about when and how those changes would truly be beneficial became more difficult for management to make. Operational research methods used by middle management were not powerful enough for complex processes. In 1955, the digital computer came to the aid of management. Middle managers now had the resource to conduct Operations Research techniques quickly. From these methods emerged another technique. The new tool was simulation. (Emshoff & Sisson, 1970)

At first simulation was not highly respected as a management tool due to its probabilistic nature. However, the advantages of simulation eventually convinced management of its usefulness. Simulation's major advantage is its cost effectiveness. If a simulation model is properly built, the effects of expansion and change can be seen before any capital is laid out for the project. Management can have the results of a project in hand to help make the decision of the projects fate.

Simulation is also a potentially dangerous tool for management. A valid model requires proper assumptions, programming, and correct interpretation of the results. Most simulation models are stochastic in nature. Therefore, a good understanding of probability and statistics is essential for proper interpretation of the results.

The history of simulation has paralleled the evolution of computer hardware and software. Personal computers (PC's) have given the scientist and engineer the opportunity to conduct sophisticated simulations for minimal expense. Simulationist's continue to demand better and more sophisticated hardware and software tools to help them in their analysis of complex systems.

2.2 Simulation languages

Simulation was first programmed using common programming languages. Primarily, the language used was FORTRAN. Eventually, computing languages were designed specifically for simulation. As industry needs grew more complex, different simulation languages were developed for different types of models.

In general,

> "simulation is a numerical technique for conducting experiments on a digital computer, which involves certain types of mathematical and logical relationships necessary to describe the behavior and structure of a complex real-world system over extended periods of time." (Naylor et. al., 1966, pg., 3)

There are many simulation languages and simulators available to assist in modeling. Selecting the appropriate tools for the job can be sometimes difficult and references to trade journals and simulation textbooks can sometimes help (Modern Materials Handling, 9/87). Some of the most popular simulation languages and simulation practitioners are listed as follows.

GPSS (General Purpose Systems Simulator) was one of the first major simulation languages. It was designed by G. Grudden for IBM. GPSS is a very general language used mostly for queuing problems and inventory control. GPSS/PC and GPSS/H are two modern derivatives of GPSS. (Gordon, 1969) (IBM, 1970) (Schriber, 1972)

SIMSCRIPT, developed by H. M. Markowitz (working with P.J. Kiviat and R. Villanueva) at the RAND Corporation and then at CACI (Simscript II.5, 1983, 1988), was the next major language. SIMSCRIPT is a powerful general-purpose programming language. It is primarily used to program discrete-event simulation models but it also allows for the process orientation approach. Simscript is provided in its own language and is available at the PC level.

A. Alan B. Pritsker has made several contributions to the advancement of simulation. The first simulation language written by Pritsker was GASP (General All-Purpose Simulation Programmer). GASP had its origins at U.S. steel and Pritsker improved it in 1969. Pritsker and P.J. Kiviat documented GASP II and then Pritsker produced the latest version GASP IV (Pritsker, 1974). It is largely based on FORTRAN and requires the user to write a series of subroutines to describe events. Pritsker's next project was the development of SLAM (Simulation Language for Alternative Modeling) which was developed with Pegden (Pritsker & Pegden, 1979). This language is a very powerful simulation language offering not only network orientation, but also discrete event orientation, continuous, or a combination of both.

Dennis Pegden, a student of Pritsker, developed SIMAN (SIMulation ANalysis) to model discrete-event, continuous, and combined continuous discrete-event systems. SIMAN utilizes block diagrams to describe the flow of entities through a system. Fishman describes this as the process interaction approach (Fishman, 1978).

Although these special purpose programming languages were an improvement over creating a simulation model from some general purpose programming language such as FORTRAN, they were limited in the flexibility that they provided. Programmers had to drop down to some base language if they wanted to do something that was not provided in the special purpose programming language. This made

some simulation projects frustrating and tended to limit the variety of simulation projects that were undertaken.

Simulationist's are demanding better and easier simulation tools. Simulation languages must therefore continue to provide the basic simulation constructs that users have come to expect and also include new tools that provide the ability to conduct better, smarter, and more flexible simulations.

AI scientists have for years provided tools that can create expert systems, neural networks, and advanced robotic control. Their inclusion into simulation models would have a profound impact on simulation modeling and analysis. Unfortunately, present day simulation languages are not designed for flexibility and the inclusion of AI tools into simulation models.

Simulation languages must therefore continue to evolve to meet the needs of the systems they analyze. The addition of new components into the simulation toolbox must continue!

2.3 Components and organization of computer simulations

Most literature on this topic categorize a simulation study into 6 elements. These elements are:

Table 1. Elements of a simulation analysis.

1.	Formulation of the problem
2.	Collection and analysis of real world data
3.	Model development
4.	Verification and validation
5.	Design of simulation experiments and optimization
6.	Implementation

In formulating the problem, we define the questions for which answers are sought, variables and parameters involved, and measures of system performance. Collection and analysis of real world data is necessary to mathematically represent the model. Model development involves selecting a computer programming language, building and testing a computer model of the real system, and debugging it. The verification and validation element or step, checks if the model is a accurate representation of the real system. Design of simulation experiments and optimization define the replications, accuracy, and experiment to be run. Implementation, the final element is important since we have spent time, money and resources on a simulation analysis. Just like the formulation of linear programming problems in Operations Research, simulation is an iterative process.

Before a simulation model is developed, the programmer or analyst selects or creates a simulation language or engine capable of meeting the needs of the study being undertaken. This language or tools,

must provide the basic components used in a simulation analysis (Table 2). It should also be flexible enough to allow for the addition of real world aspects into the simulation model.

Table 2. Components of computer simulation languages.

1.	Time flow mechanisms
2.	Integration abilities
3.	An event calendar or event list
4.	Random number generators
5.	Statistical gathering capability
6.	Experiment packaging
7.	Reports

There are two types of time advance mechanisms used in computer simulations; 'Fixed Time Increment', and 'Variable Time Increment'. Fixed time increment advance is said to be more efficient for systems where events occur at regular time intervals, whereas variable time increment is said to be more efficient for systems where events occur at variable time intervals (Blake, et. al., 1964, pp. 14-20). In addition, variable time increment is said to save computer time when a simulation is static for long periods of time. In the variable time increment method, time is treated as a continuous variable. Most simulations are conducted using variable time increment, thus, fixed time increment simulations will not be further addressed.

In the variable time increment method, the simulation clock is initialized to the starting time for the simulation and registered events are placed on the event list. The event list is a list of future events which is sorted from top to bottom by ascending time and ascending priority. When the simulation begins, the clock removes the first event on the event list, updates the current time to the event time, and then updates the event. When the system is updated in the event routine, new events can be scheduled and added to the event list. The process repeats by removing the next object at the top of the event list, resetting the clock, and then updating the object. The procedure of advancing the simulation clock continues until no more items are on the event list, or some pre specified stopping condition ends the simulation. Variable time increment simulations therefore jump through time from one event to the next event, and the state of the system is updated during that event. Figure 2 displays an event list and graphs the times when these events will occur.

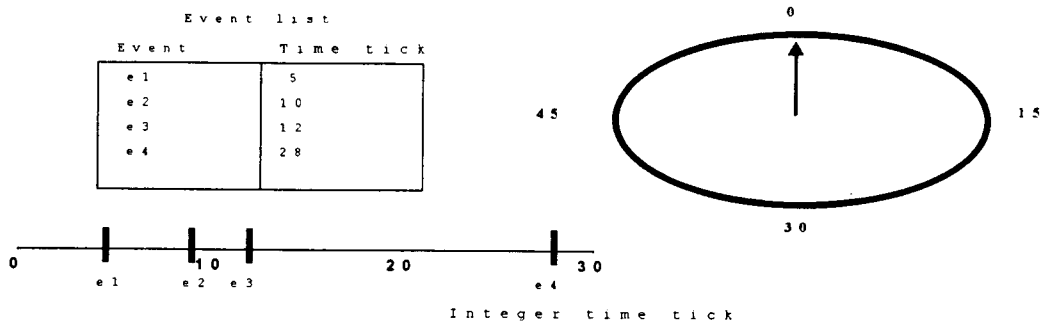


Figure 2. Variable time increment execution.

In summary, a simulation begins at some starting time, usually 0, by initializing the clock, initializing statistics, and initializing the event list. The clock removes the first item on the event list, adjusts the clock time, updates the event, and then repeats. During the updating of the system, new events may be registered. The simulation will continue indefinitely until the event list becomes empty or some pre specified condition stops the simulation. Figure 3 presents a flow chart of the simulation process.

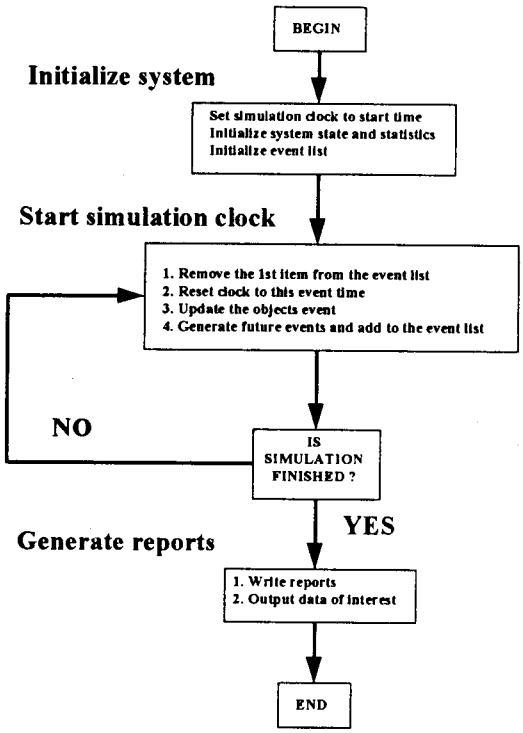


Figure 3. A flow chart of the simulation process.

In table 2, item two, "integration ability" is another important component of a simulation language. Differential equations are fundamentally important in engineering because many physical laws and relations appear mathematically in the form of such equations. Although there are numerous methods for solving differential equations, most involve numerical methods. Differential equations are equations of derivatives. Derivatives define rates of change of some variable over another, usually time or space. If all the derivatives are taken with respect to the same variable, the differential equation is said to be ordinary.

Most simulations that employ differential equations use ordinary differential equations (ODE's) with respect to time. Partial differential equations (PDE's) are derivatives which define a rate of change over more than one variable. Heat transfer problems can define rates of change over time, and three dimensional space. ODE's can be first order, second order, or any higher order required by the derivative. Fortunately, all higher ODE's can be reduced to a set of first order differential equations. Computer programs that solve differential equations generally require a set of first order differential equations. The solution to first order differential equations is usually specified by the nature of the boundary conditions for the variables being integrated.

Boundary conditions divide into two categories, the initial value problems, and the two-point boundary value problems. Initial value problems are when some variable or 'y' value is given some starting value and then integrated over some variable like time. Two-point boundary value problems are when some variable is given a starting and ending value. Two-point boundary value problems are generally more difficult than initial value problems and their discussion will not be further addressed. It should be noted that many two-point boundary value problems can be rewritten as initial value problems (Numerical recipes in C, 1988).

The three most popular methods of solving ODE's are:

- Runge-Kutta methods
- Bulirsch-Stoer methods
- Predictor-Corrector methods

Runge-Kutta methods utilize Taylor type polynomials to approximate a polynomial up to some high order. The advantage of Runge-Kutta over other methods of approximating Taylor polynomials is that Runge-Kutta doesn't require taking derivatives. Runge-Kutta uses functional values to substitute for derivative information while integrating over the interval.

Bulirsch-Stoer methods utilize rational functional extrapolation and midpoint methods to integrate a variable. They can obtain high-accuracy solutions to ODE's with minimum computational effort.

Predictor-Corrector methods store a solution, extrapolate across a step, and then correct the step using derivative information at the new point. Predictor-Corrector methods are for high precision very smooth functions.

If we refer back to table 2, item 4, we see that random number generators are another important part of any computer simulation languages. Table 3 lists the minimum requirements for any random number generator.

Table 3. Minimum requirements of a random number generator.

1.	It should generate repeatable sequences
2.	The code should be fast and only use a small amount of memory
3.	The uniform variates should pass reasonable tests for being from the uniform distribution
4.	The uniform variates should pass a number of tests for randomness
5.	The lattice structure should not be obviously present
6.	It should be portable

Prime Modulus Multiplicative Congruential Generators (PMMCG) are the most popular and best studied generators available today. The form of the equation is;

$$X_i = (a * X_{i-1}) \bmod m, \tag{1}$$

where m is prime, and a is a primitive element modulo m. If a and m are chosen such that the smallest integer b for which a^b - 1 is divisible by m is b = m - 1 (Knuth 1981, p., 19), then we can obtain every integer from 1, 2, ..., m-1 exactly once in each cycle. Thus (Xi-1) can be any integer from 1 through m-1 and a full period of m-1 will result. Using m = 2^31 -1 (2,147,483,647) and a as a primitive element modulo m, will produce the maximum cycle length possible with 32 bit integer arithmetic. Initial seed values can be any integer value from 1 to 2^31-2. A value of 0 or 2^31-1 will produce an infinite sequence of zeros. Once the generator is started with some initial integer seed value, it will eventually return every integer value between 1 and 2^31-2 exactly once before the cycle repeats. This will allow for more than 2 billion unique random numbers for each stream.

Fishman and Moore have proposed 5 optimal full period multipliers for the PMMCG with prime modulus 2^31-1. They present a battery of statistical tests for evidence of their statistical robustness (Fishman and Moore, 1982). They also discuss poor results of some previously used multipliers (Fishman and Moore, 1986). Some of the most popular multipliers are listed in Appendix H.

Another component listed in table 2, was the ability to collect statistics. Since most simulations are just computer based statistical sampling experiments, the ability to gather statistics is a necessary

requirement of every simulation language. If the results of a stochastic simulation experiment are to have any meaning, appropriate statistical techniques must be used to design and analyze the simulation experiments. There are basically four types of statistics which are minimum requirement for any simulation practitioner; Discrete Time based, Continuous Time Based, Observational based, and Counting. Discrete and Continuous Time based statistics are both statistics that are based on the fraction of time a statistic is at some value. The difference between continuous and discrete time based statistics is that the continuous statistic is continuously changing over time and the discrete statistic is changing in finite discrete jumps over time. Observational based statistics are discrete in that they are generated over some counting value. Count statistics are a discrete statistic generated by counting some occurrence. Taha (1988) provides a good explanation of all but continuous-time based statistics. Appendix I details how the different types of statistics are collected.

Finally, a simulation language would not be complete if it couldn't generate reports. The generation of reports at the completion of a simulation is a necessary component of any simulation analysis. Reports should provide general information about the simulation model and experiments that were run. In addition, reports should output statistical summaries and random number generator summaries. Single run reports should summarize the single run whereas multiple runs (replications) should be summarized by a global report summary. Example reports are shown in Appendix E for a discrete and continuous simulation.

A complete simulation report is a document which presents the results of a simulation study (McLeod, 1982). A complete simulation report should contain at a minimum, the following components (McHaney, 1991):

Table 4. Minimum requirements of a simulation report.

1. Title page
2. Table of contents
3. Statement of objectives
4. Methodology
5. Conclusions and recommendations
6. Findings
7. Appendix

An added feature of many simulation languages and environments is the built in capability of automatic report generation. Standard reports are generated which list the random number distributions used, statistical observations collected, and general experiment settings. Most simulation languages allow the generation of user specified reports at the end of a simulation. SIMAN calls the user defined 'WRAPUP' function at the end of a simulation (SIMAN, 1990). This gives the programmer the ability to

easily generate any special reports at the end of the simulation. Most other simulation languages have a similar system or allow the user to write their own report method.

2.4 Object oriented simulation

Simulation languages evolved from ALGOL and FORTRAN, introduced in the 1950's, and provided the ability to code complex mathematical expressions, to GASP and SIMSCRIPT, introduced in the early 1960's, to Simula, introduced in the late 1960's, which provided the class concept, to languages of the 70's: Pascal, GPSS, and SLAM. Ada was introduced in the late 70's, C++ was added in the early 80's. GPSS and SLAM are FORTRAN based simulation languages which added simulation constructs to aid in simulation development. Items like; the simulation clock, the event list, random number generators, statistical gathering, and means to monitor simulation progress work, came to be considered minimum component tools for a computer simulation language.

The history of object oriented simulation (OOS) goes back to the creation of Simula (Dahl and Nygaard, 1966, 1967, 1967). Simula was the not only the first simulation language but it was also the first object oriented language. Simula was the first to introduce the class concept, a necessary requirement of OOP. Simula never gained widespread use in commercial environments but it did have a strong academic following in Europe as well as great influence in programming language design. In addition to basic language features, Simula provides simulation support through two key simulation classes: class SimSet and class Simulation (Franta, W. R., 1977). Class Simset provides list processing by implementing a doubly linked list. The event list and all types of queues use class SimSet. Class Simulation provides scheduling, synchronization, and a simulated time flow mechanism.

There have been many implementations of OOS languages. DEMOS (Birtwistle, 1979), consists of 3000 lines of Simula code designed to model queuing networks. DISCO, a further extension to DEMOS which provides classes to assist in combined simulations (Helsgaun, 1980), demonstrated how easy it was to extend OOS languages like Simula. The Smalltalk "blue book" (Goldberg, 1983) explains the simulation constructs that were implemented in Smalltalk-80. The design of Smalltalk was strongly influenced by Simula. MODSIM II, based on Modula-2, is an object oriented general purpose programming language which provides direct support for OOP and discrete event simulation (Belanger et. al, 1990). C with classes was introduced in 1980 (Stroustrup, 1983), and C++ since the summer of 1983 (Stroustrup, 1986). C++ was originally developed because Stroustrup wanted to write event-driven simulations, like ones written in Simula '67', but with the efficiency of C. The C++ programming language could be described as "C++ = C + Simula".

Considering the history of OOP and C++, it's not surprising that C++ can be an effective simulation language. Simulation is one of the most natural applications of OOP. Languages like FORTRAN are poor cognitive tools, C++ and OOP languages are good cognitive tools because they let the program designer

concentrate on abstract concepts which are implemented as abstract data types (class). FORTRAN is a poor cognitive language because it is highly procedural and requires the programmer to think in terms of program flow. OOP languages offer a more natural representation for program coding by allowing the creation of objects. Objects encapsulate a name, structure, and behavior into one unit. Programming flow is formed by communication among objects.

OOS languages don't provide everything that the simulationist would want. Two important features that are not standard in OOS languages are inter-object communication without object dependence, and intelligent simulation capabilities. Inter-object communication without object dependence is highly desirable and not a built in feature of any language. Programmers and simulationist's have long enjoyed AI techniques but incorporating them into a simulation model has never been simple.

2.5 Discussion

As automated systems become more complex, the need for stochastic simulation analysis is increasing. Analytic models are absolutely best when they can be applied but a typical automated system includes multiple queues, prioritizing, arrival and service distributions, and interaction. Special purpose simulation languages have simplified the complexities of modeling automated systems. SLAM (Pritsker 1979), SIMAN (Pegden 1981), and GPSS/H (Schriber, 1990) are presently the most all-inclusive simulation languages. They allow discrete-event, continuous, and combined continuous discrete-event models. Since single simulation runs produce point estimates of particular measures, multiple runs are necessary for statistically valid results.

Artificial intelligence tools like neural networks and expert systems will become additional tools added to the simulationist's toolbox. Graphical user interfaces (GUI's) and object oriented data bases will be new items added to the minimum requirements for a simulation language. Object oriented languages hold the greatest promise to the development and improvement of simulation tools and languages because they are powerful and flexible. Simulation code that is developed with an OOP language can be easily modified or extended to meet the users need.

It should be obvious that while present day simulation tools provide much, they are deeply in need of added capabilities. Two items that should be added to all simulation languages are; inter-object or inter-process communication and intelligent simulation capabilities. Different simulation objects or processes would benefit from some form of independent communication. Adding AI tools like expert system capability directly into a simulation language would create a more autonomous simulation environment. Therefore, a simulation language that provides a core simulation engine with all the simulation constructs users have come to expect, and added features like inter-object communication and intelligent simulation capabilities will fulfill a great need to society.

3. SIMULATION OBJECTS

In an object oriented simulation language, simulation objects are simulated through time. These objects may come in different shapes and sizes but they all know how to update themselves through time. In figure 4, the different shapes represent different simulation objects. These simulation objects all have state variables that start with some initial value at the start of the simulation, and are then updated as simulation time progresses.

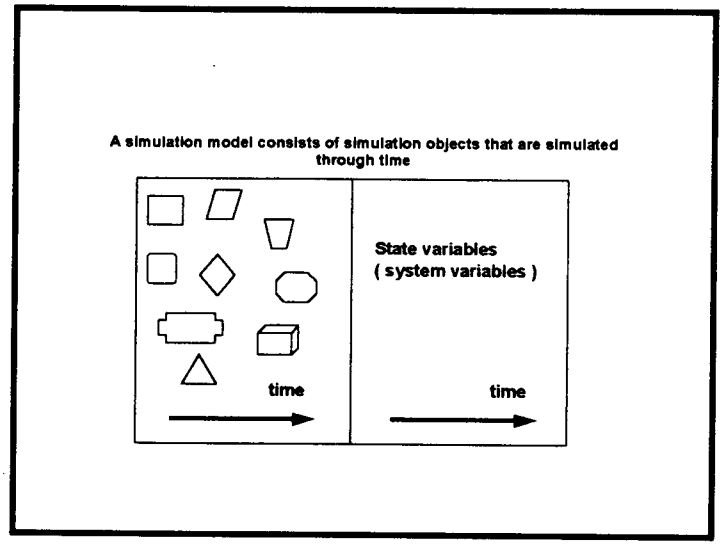


Figure 4. Simulation objects come in different sizes and shapes.

3.1 The SimObject and StatSimObject abstract classes

The simulation object classes SimObject and StatSimObject provide the abstract descriptions, random number generator requests, statistical gathering requests, output data requests, and tracing requests for all child simulation objects derived from them. SimObject is the parent class of StatSimObject. Simulation objects can be derived from either class although stochastic capabilities are provided by StatSimObject. Simulations that don't require the stochastic element can derive simulation objects directly from class SimObject. Class SimObject provides data and behavior that is common to all simulation objects and StatSimObj supplements stochastic features (Figure 5).

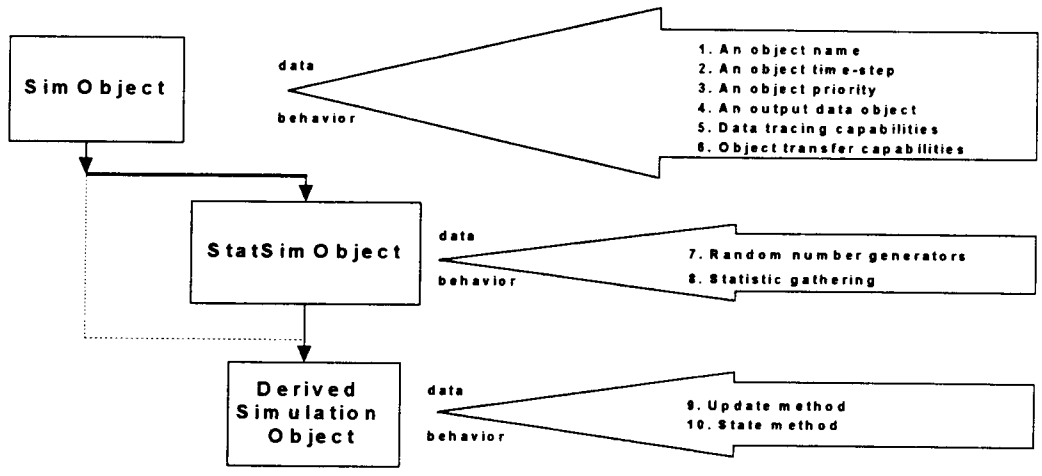


Figure 5. The simulation object hierarchy.

A simulation object is a class derived from the abstract classes SimObj or StatSimObj. A simulation object has data and methods just like any object. Its data and methods are specifically designed to aid in the simulation of the object itself. These simulation objects provide the data and behavior specific to their uniqueness while the parent class SimObject provides data and functionality that is common to all simulation objects. A simulation object called Population, a Widget generator, may provide specific details like how the simulation object is updated during a simulation event but utilize data and functionality from the parent class SimObj. Simulation objects are designed to package data and behavior that is representative of the object being simulated. Descriptions of the two abstract classes follow. Source code is provided in Appendix F.

In figure 5, the class SimObject provides certain data and behavior to a derived simulation object. Every simulation object has a name. This name is a character string and it's purpose is to identify the object with a meaningful name during reporting, debugging, and tracing. An objects time-step is the time interval between object updates. Deterministic simulation objects are updated at a constant interval while stochastic simulation objects are updated at intervals established by their random number distributions. The priority of a simulation object establishes it's ordering sequence in the simulation event list when two objects are being updated at the same time. GUI objects that show the state of a simulation object should be updated after the simulation object. This is accomplished by giving the GUI object lower (higher priority value) priority.

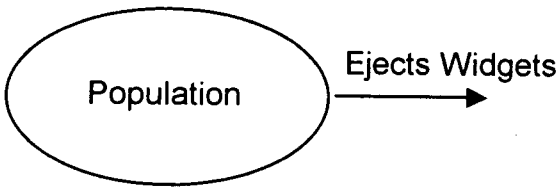
It is sometimes useful to have simulation objects store data. This storage of information can be accomplished by giving a SimObject a reference to a DataObject. The DataObject is a repository for data. The ability to debug a simulation object is another necessary requirement of simulation languages.

Tracing lets a user view the state of simulation objects through time. The TraceArray is an array of structures that store information about the current state of an object. The simulation objects trace can be directed to an output device for debugging purposes. StartTrace, StopTrace, and InspectProperty are the interface to tracing the state of a simulation object.

One thing that is common in many simulation languages, is the ability to transfer tokens or objects from one object to another. In a queuing simulation, customers in a bank can be considered a Widget or Token that is passed from one object in the system to the next object in the system. Also, manufacturing lines that produce some product will move a component from one workstation to the next. Thus, object transfer capabilities are considered necessary in a simulation language.

To provide such object transfer capabilities, the SimObject class provides two mechanisms to transfer objects between objects. The first method is a Transfer function which requires a reference to the receiving object. The second method is actually provided with the assistance of the simulation environment through the Broadcast-Notify and Broadcast-Notify-Transfer methodologies. These message and object passing schemes have proven very useful because the sender doesn't need to know who the receiver is. Chapter 4 discusses the useful message passing schemes available through inter-object communication.

If we refer back to figure 5 again, we see that the StatSimObject provides access to two important simulation components; random numbers and statistical gathering. Random numbers are made available to simulation objects that are derived from class StatSimObject. StatSimObject provides access to eighteen different random number generator classes. Simulation objects that are stochastic in nature typically use random numbers to set their update time-step. In figure 6, the derived simulation object "Population" generates one Widget object every 'timeStep'. The 'timeStep' is assigned a random value from some user specified distributed.

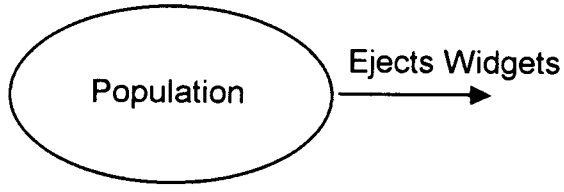



```
timeStep = pRand->RandValue();
```

Figure 6. A derived simulation object using random numbers to set its Update time-step.

Simulation objects can collect their own statistics as they are simulated through time. There are four statistical collection classes provided by the simulation engine. Access to the statistical collection classes

is provided through the StatSimObject class. In figure 7, the derived simulation object "Population" collects statistics on the number of Widgets produced during the simulation.




```
AddCountObservation( ID_WIDGET_COUNT, 1 );
```

Figure 7. Recording a statistical observation.

In every simulation language, the state variables of a system can change value only during an event. Thus, "an event is defined as a change of state". Changes in state can occur continuously as time evolves or at discrete moments in time. If a change of state is continuous over time, then integration procedures must be utilized to change the state variables. Discrete changes in state variables do not require integration procedures and are therefore much simpler since no ordinary differential equations are required.

State is a method of a simulation object which packages the differential equations necessary to update the state of the simulation object. Update is also a method of a simulation object which defines the actions to be taken to update the simulation object. Update is the method that is called when the simulation object needs to be updated. Several authors use the terminology 'Event routine:' (Law & Kelton, 1991), (Kiviat et al, 1968), and (Pritsker et., al., 1969), but the use of the word 'Update' is more appropriate in the OOP context.

Update and State are two methods of the simulation objects that define how the objects state changes. Update is utilized in both discrete and continuous simulation objects, while State is only utilized in continuous simulation objects. Update is where the simulation object packages the logic that handles the state change of the object. Continuous simulation objects that have differential equations use Update in concert with the numerical integrator and the State method to integrate and update the simulation object over the 'timeStep'. Examples of the Update and State methods are given in the multi-server queuing system and bog ecosystem (Appendix E), and figure 8. The Update method for a derived simulation object is called automatically when an event is removed from the top of the simulation event list.

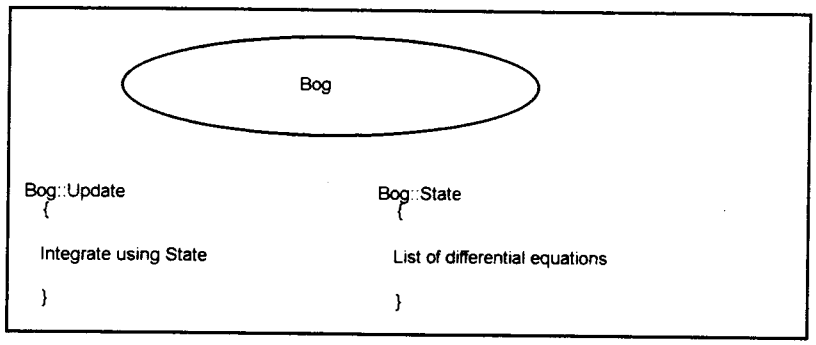


Figure 8. The Update and State methods for the Bog simulation object.

3.2 Derived generic classes

The derived generic simulation classes are simulation objects that can be re-used in many simulation models. The Population class, Queue class, ResourceServer class, Resource class, and Sink class have direct applicability in queuing models but practitioners should not limit their extendibility. Ecosystem simulations that have sources, sinks, producers, and consumers, can also use the generic simulation object templates as a starting point to the development of their simulation objects.

Since SimObject and StatSimObject are abstract simulation objects, they cannot be directly instantiated. The derived generic simulation classes in figure 9 show how the a programmer could organize a simulation object hierarchy.

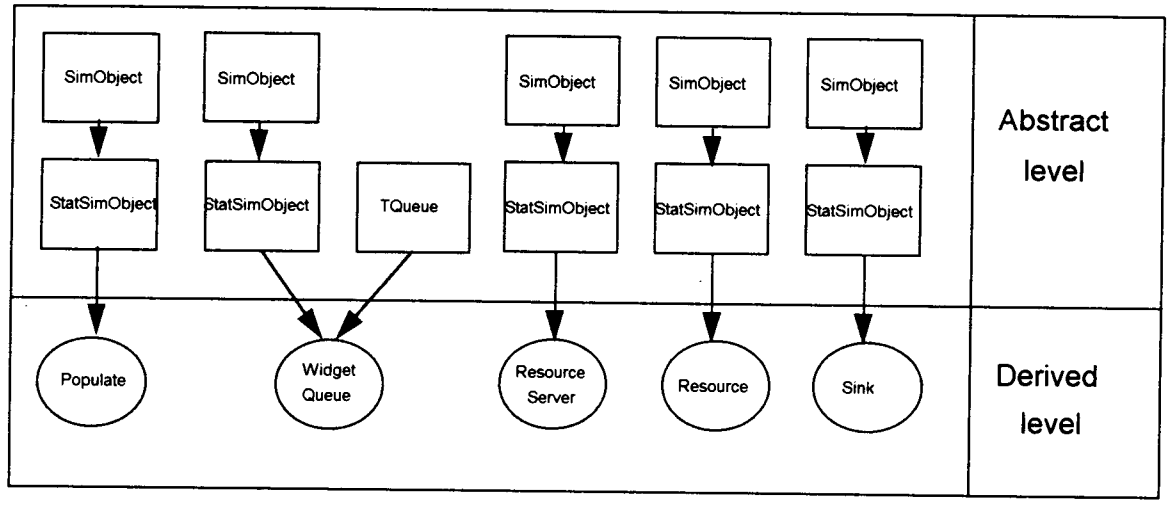


Figure 9. The derived generic simulation classes.

The generic simulation classes are examples of simulation objects that can be used as templates for the creation of new simulation objects. They also have direct applicability in queuing models. Section 7.2 and Appendix E show the generic simulation classes being used in a multi-server queuing system.

There are many details about the simulation object class hierarchy that would make this discussion long and complex. In order to limit the topic, the reader is encouraged to examine the examples provided in Chapter 7 and Appendix E. Programmers will want to investigate the source code provided in Appendix D.

4. INTER-OBJECT COMMUNICATION

Communication among objects is what makes object oriented programs useful. Objects send messages to other objects to implement some action or get some information. In figure 10, object 1 sends a message to object 2. Object 2 takes some action in response to the message being sent, and replies to the sender of the message. This is a form of object dependent communication because object 1 must have some reference to object 2 to send it a message.

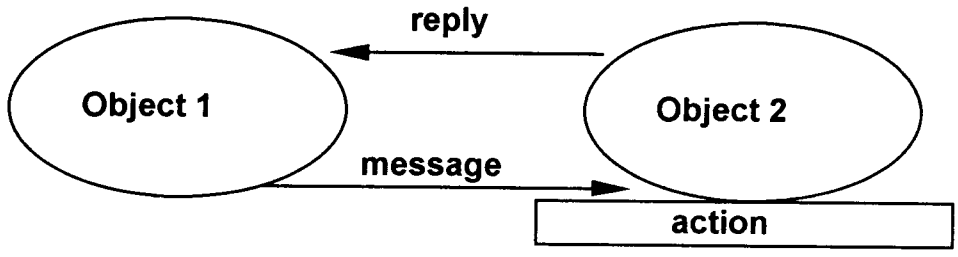


Figure 10. Dependent object communication.

Message passing is nothing new to an object oriented programmer. Programs are created by building objects and sending messages. The problem with this type of communication is that it introduces object dependencies. In figure 10, Object 1 must have some reference to object 2 to send it a message. Object dependence can become a problem in simulation modeling. There are situations where the object sender may not know who the object receivers are. In figure 11, the 'Ice-cream man' Broadcasts to anyone within hearing range that it has ice-cream to sell. Anyone interested will receive the message and respond accordingly. What is needed in this situation is some form of object independent communication.

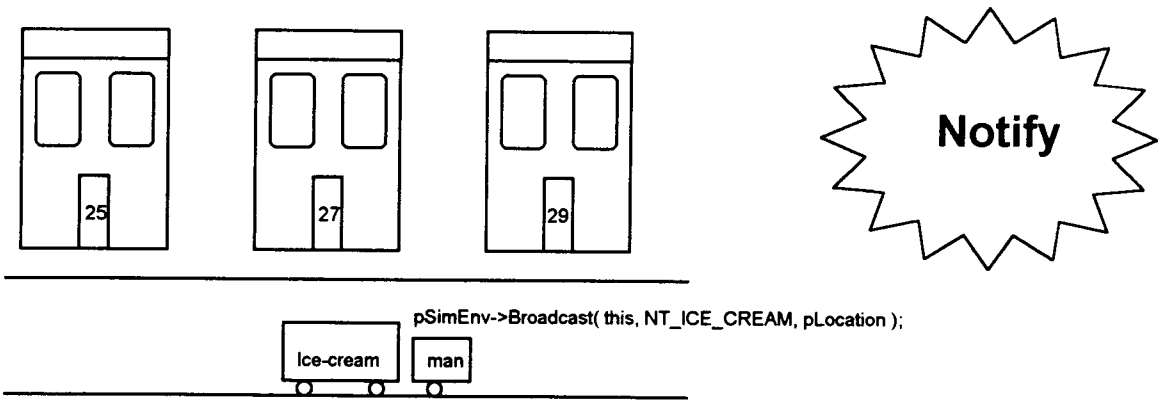


Figure 11. Ice-cream man Broadcast's a message.

During the development of the simulation environment, it was found that some form of independent object communication was needed. There needed to be some protocol available so that a simulation object could send messages to other objects in the system without requiring a direct reference to them. The implementation is quite simple. First, an object must tell the simulation environment what messages it is interested in. Second, an object must Broadcast messages to the simulation environment when it wants other objects in the system to hear those messages. When an object adds a notification filter to the simulation environment, it says that it wants to be notified whenever a particular message is Broadcast from certain simulation objects. To transmit some message, an object Broadcast's a message to the simulation environment, which will then Notify all objects that have previously registered interest in such messages. This form of object independent communication can be named after the methods that are used to implement it. The Broadcast-Notify and Broadcast-Notify-Transfer methodologies handle the independent communication among simulation objects in the simulation environment system.

4.1 Adding notification filters

AddNotifyFilter is a way for an object to say "I'm interested in being notified from these simulation objects when this event occurs". Notifications are unique, and the number of simulation objects in the notify filter can be none, one, or as many as every simulation object that is registered with the simulation environment. AddNotifyFilter works with Broadcast to provide an additional form of communication between objects. For example, an object Broadcast's that it wants to transfer some data. Any object that has registered interest in this type of notification from this simulation object may have an opportunity to receive this data. The order of opportunity in receiving such a message is linked to the ordering of simulation objects in the simObjList.

The first step in object independent communication is for objects to add notification filters. When an object adds a notification filter to the system it is saying that it is interested in some messages from some

simulation objects which are part of the simulation system. Figure 12 shows a simulation system made up of five simulation objects. The simulation object 'SimObject' wants to tell the simulation environment that it is interested in certain messages from certain simulation objects. Specifically, 'SimObject' is interested in receiving help (NT_HELP) from any object in the system, accepting transfers from Object A, B, or C, and having the Expert answer its questions.

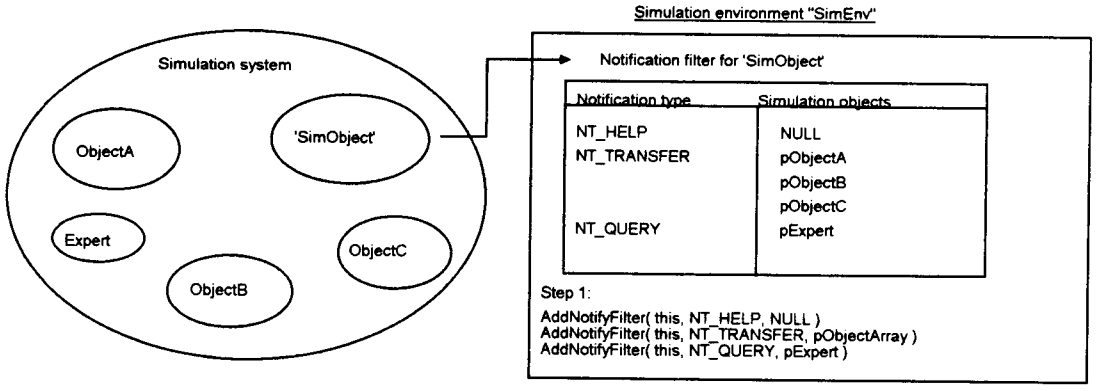


Figure 12. A simulation object adding notification filters.

4.2 Broadcasting messages and transferring objects

The second step in object independent communication is for objects to Broadcast their messages. When an object Broadcast's a message to the system, the system takes the message and routes it to objects that have previously expressed interest in this message from this object. Figure 13 shows the inter-object communication process.

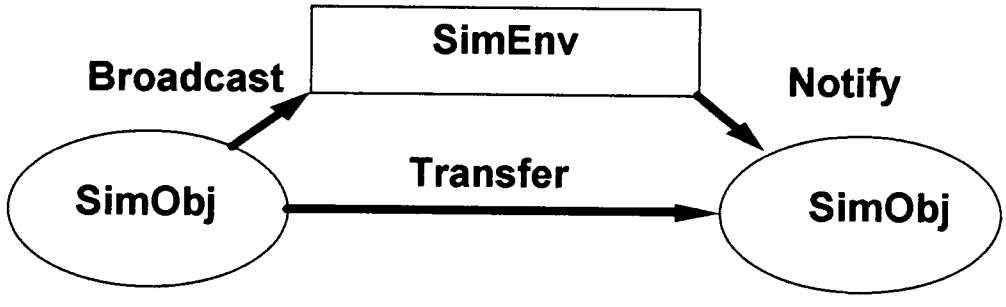


Figure 13. Inter-object communication with Broadcast-Notify-Transfer.

In addition to sending notification messages to other objects, an object can send a notification message that it wants to transfer a particular type of object. Objects can Broadcast a message like 'NT_TRANSFER' and include in the argument list, the object it wants to transfer. This can be described as the PUSH scenario because we push an object to another object by using the Broadcast message. Objects that have registered interest in the Broadcasting object will get the opportunity to receive this Broadcast. If the receiver of the Broadcast message wants to accept the transfer of the object, it replies TRUE to the broadcasting object and takes the sent object. If the receiver of the message is not interested in the message, it replies FALSE. When all message receivers respond FALSE to a message senders request to transfer an object, then it is the responsibility of the message sender to handle the disposal, balking, or alternative transfer of this object.

Another way to transfer objects could be called the PULL scenario. Objects can Broadcast a NT_INIT_TRANSFER message when they want to receive a certain type of object. Objects that have expressed interest in receiving messages from this object have the opportunity to respond with a TRUE or FALSE. If an object responds TRUE to this message, it should place a reference to the object to be transferred in the extra data field of the notification filter. This process is very similar to the PUSH scenario except that objects are pulled from the object that receives the message instead of being pushed to the object that receives the message. If the response is TRUE, the sender object knows that the extra data field of the Broadcast message will contain the object which was pulled from the object that received the message.

The notification scheme and transfer of objects is fully demonstrated in the multi-server queuing system (Appendix E).

4.3 The bulletin board

The bulletin board is an object that can be used by any other object. It was first introduced by AI scientists (Charniak and McDermott, 1985). The bulletin board is a place where objects can post messages. Usually, simulation objects will post messages to the bulletin board, and Expert simulation objects will read these messages. The process of passing and reading messages can also be considered a form of inter-object communication. The difference between the bulletin boards form of communication and the Broadcast message passing scheme is in the speed of delivery. Broadcasted messages are sent in real time whereas messages pulled from the bulletin board occur only at events. The amount of time between a message being posted to the bulletin board and being read is unspecified. Figure 14 shows the bulletin board which has been implemented as a list of structures.

Bulletin Board

messege ID	time	posting object
TAG_1	0	pObjectA
TAG_5	11	pObjectA
TAG_38	38	pObjectC
TAG_2	101	pObjectD

Figure 14. The bulletin board object.

Simulation objects communicate with other simulation objects via the Broadcast-Notify, Broadcast-Notify-Transfer, and bulletin board methodologies. These types of communication do not require that the sender of a message know the receiver. Broadcasting and posting to the bulletin board are two powerful message passing schemes which allow for global or object specific communication without introducing object dependencies. While the bulletin board idea is not a new concept, the Broadcast-Notify and Broadcast-Notify-Transfer methodologies are new and very important to the flexibility of the simulation environment.

5. CORE SIMULATION ENGINE

There are several classes which when working together, make up a simulation engine useful for simulation modeling and analysis. The simulation environment classes SimEnv and StatSimEnv largely handle the clock, event list, communication, replications, and reports. The Rand class provides random number generators from eighteen different distributions. The Statistical collection classes StatsObs, StatsTime, StatsContin, and Counter provide all the statistical gathering features required for every type of simulation. Global statistics can be produced and made available by the ReplicationStats class. Numerical integration techniques are provided by fourth order Runge-Kutta and Runge-Kutta Feldberg. A complete description of these classes follows.

5.1 The SimEnv and StatSimEnv simulation environment classes

The simulation environment classes SimEnv and StatSimEnv provide many of the necessary components for simulation modeling. SimEnv is the parent class which provides a simulation clock, event list, object list, and standard reporting capabilities while StatSimEnv, the child class provides replication features and global statistics. The classes have been separated because deterministic simulations do not require replication features (Figure 15).

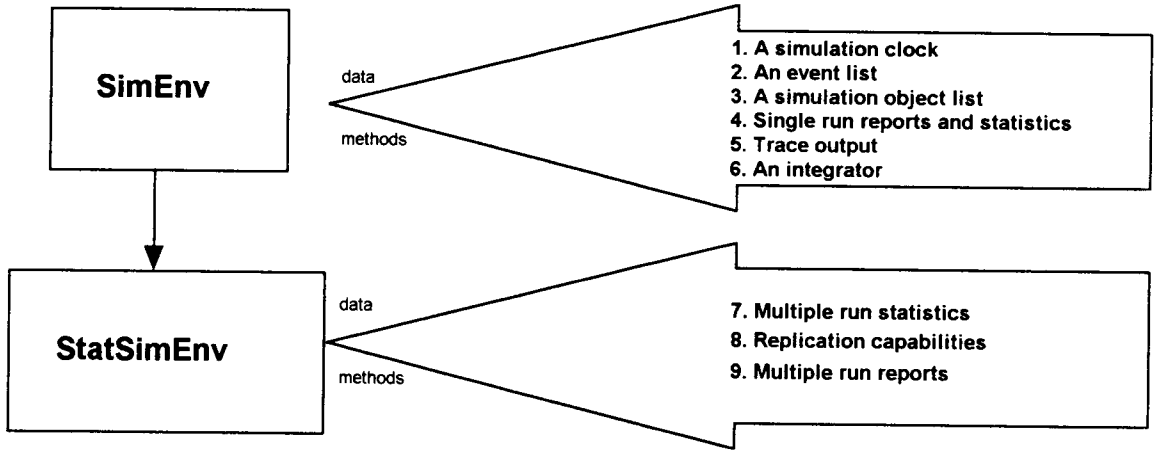


Figure 15. The simulation environment classes.

SimEnv is a class, and an object or instantiation of this class provides a timing mechanism or simulation clock and basic simulation features common to simulation engines for computer simulation. Like all objects, SimEnv has data and methods. The data is protected, as it should be for all well designed

objects, and public methods are provided to allow messages to be passed to the SimEnv object. Likewise, StatSimEnv is a class and it is packaged in the same format as SimEnv. Descriptions of the two classes follow. Source code for the two classes is provided in Appendix G.

The behavior of the SimEnv class is specified in its methods. SimEnv is the constructor for the class. SimEnv initializes itself and provides through its methods, clock and event list capabilities for a discrete, continuous, or combined simulation.

Clock control is implemented through get and set methods like; SetStartTime, GetStartTime, SetTimeStep, GetTimeStep, SetStopTime, GetStopTime, ResetCumTime, GetCumTime, and GetCurrentTime. These methods provide access to SimEnv data which controls its behavior.

RegisterSimObj is a method that places pointers to simulation objects in the simObjList. It is necessary to register simulation objects with a SimEnv if events from a simulation object are to be placed on the event list. One would normally register a simulation object with automatic updating if they were conducting a deterministic continuous simulation. If automatic updating was not selected, then one would typically register events for that simulation object. Section 5.1 and 5.2 provide details on the simObjList and simEventList.

RegisterEvent is probably the most useful method in SimEnv. Since events are what drive the simulation, a simulation with this environment will not take place unless events are registered. An event on the event list is a structure which contains; a SimObj, an eventIncr or floating point time in the future when this event shall occur, a priority where 0 is the highest priority and 100 is the lowest, an eventType, and a void **pointer for any extra data. Registering events can take place at anytime during a simulation because the event always takes place eventIncr time in the future. The time in the future when the event will occur can be any floating point value from zero to the largest 32 bit floating point value.

Simulation analysts often find the development of simulation logic complex and confusing. Two features of the SimEnv class assist the user in the understanding of simulation flow and object state. GetEventListPtr provides external access to the clocks event list. This method is useful during debugging when a user wants to see if all the proper events are showing up on the event list. Another feature called tracing, outputs the state of a simulation object through time. If a simulation object has tracing enabled, then the SimEnv class will output the state of this object as it is simulated through time.

ViewEventList allows the user to view the entire eventList for this environment. This can be a necessary requirement during debugging a simulation and it also gives the user a better understanding of the flow of events through time.

Generating reports after or during a simulation is an important part of the simulation process. Five types of statistics can be collected during the simulation and reports from the statistics can be generated by enumerating these statistics for each simulation object that has collected them. Enumeration is handled automatically by the methods EnumObsStats, EnumContinuousStats, EnumTimeStats, EnumCountStats, and numDistStats. The collection of observational, continuous-time based, discrete-time based, and

counting statistics are the responsibility of each simulation object. Distribution statistics are generated automatically from the distribution object Rand. SimEnv can loop through the simObjList and check each simulation object for any or all of these statistics and appropriately enumerate those statistics to the proper reports.

In figure 15, we see that the StatSimEnv class provides data and behavior to support replicated simulations. StatSimEnv uses one abstract data type (ADT) called ReplicationStats to record replicated statistics. The ReplicationStats class is discussed in section 5.4 and computer code is given in Appendix J.

Individual runs of a replicated simulation are controlled by the Replicate method. Replicate uses a callback function to a user defined experiment to run the independent replications. Everything about the system state, with the exception of the random number seeds, is initialized to the same initial state for each replicate.

StartRun and EndRun define pre-run and post-run processing. These methods handle the setting of random number seeds and the recording of a single run of statistics. RecordGlobalStatistics is called during EndRun and it outputs the single run statistics to the ReplicationStats object.

StatSimEnv like SimEnv provides default reports at the completion of a simulation run and after a group of replications. These reports list general information about the experiments run and list all statistics collected during the simulation. The reports were provided to users of the simulation environment as a template for user specific reports. Users will find the reports adequate for most needs but not be limited in their ability to extend the output provided in these reports. SetGlobalReportProc defines the user specified output report for the replicate simulation.

5.1.1 The simulation object list

The simulation object list is a data member of the simulation environment. The object itself is implemented as container that holds simulation objects. (Figure 16)

Simulation object list

simulation object	NotifyFilterArray
pSimObj1	pNotifyFilter
pSimObj2	pNotifyFilter
pSimObj3	pNotifyFilter
...	...

Figure 16. The simulation object list.

The simulation environment requires that simulation objects be registered with the simulation environment and that a reference to the simulation object is placed on the simObjList. The simObjList is necessary so that the simulation environment will always have access to the simulation objects that will participate in the simulation. It could be stated that the simObjList is a container for the simulation objects that make up the system being simulated. Figure 17 shows what a simulation object list might look like in the context of some system being simulated.

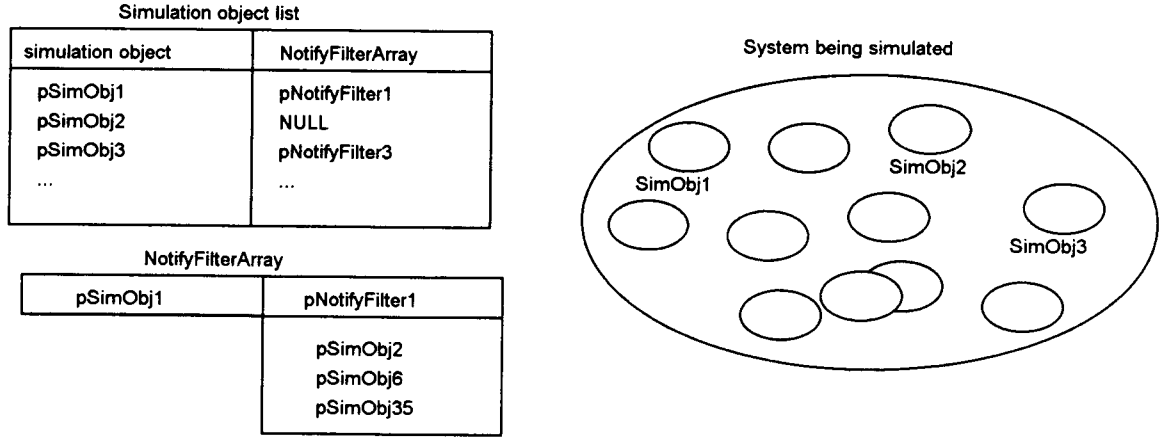


Figure 17. The simObjList for a system.

5.1.2 The simulation clock and event list

The simulation clock and event list are two components of the simulation environment that virtually run the simulation. Since events are place on the event list in increasing time, the simulation clock can update its current time each time it removes an event from the event list.

Continuous and discrete simulation objects are simulated through time by updating their state during an event. Events are therefore the activities that drive the simulation through time. An explanation of how the event list works and what is necessary to make it work will assist in understanding a principal component of the SimEnv class.

The event list is a list of events that are sorted by first time, then priority. From top to bottom on the event list, time and priority are in increasing value. When events are pulled off the top of the event list, the simulation environment extracts from the event, the simulation object to be updated and the time of that event. After the environment updates the current time, it sends a message to the simulation object to update itself. The simulation environment will continue to run until either no items are on the event list or the clock stop time has been reached.

Before events can be registered with the simulation environment, the simulation objects of the system must be registered with the simulation environment. To register simulation objects with the simulation

environment, a message is sent to SimEnv object to RegisterSimObj. RegisterSimObj specifies the simulation object to register and whether an event will be immediately registered. Simulation objects that simulate at the start of a simulation would be registered with the automatic registration flag. Simulation objects that do not begin simulating at the start of a simulation, but will have events registered in the future, will be registered with no automatic event registration. Both stochastic and deterministic simulation objects are handled exactly the same and the simulation objects timeStep specifies the time increment to its next event.

Events are added to the event list through the action of registering an event. Registering an event requires the specification of a simulation object, time increment to the event, priority of the event, and event type. There are two ways to register events, one direct, and one indirect. The direct way is to send a message to this class "SimEnv" to RegisterEvent. The indirect way is to register the simulation object with automatic event registration.

5.2 The Rand abstract random number generator class

The Rand class is a Prime Modulus Multiplicative Congruential Generator (PMMCG). Rand is an abstract class that generates a uniform zero-one (U(0,1)) random number for its children classes. The Rand class and the children classes were designed to be flexible, easily understandable, small, and somewhat efficient. Users that need more efficiency would make greater use of inline functions. Each instantiation of a random number object will be unique and the only relationship with other random number objects is in the use of the static defaultSeed variable. Rand increments default seeds by 1 million. Computer code for the 18 most popular distributions are included (Appendix H) and a description of the 18 different child classes is given in section 5.2.1. Also, Appendix H includes a program which generates 1,000 random values from each of the eighteen different distributions.

The Rand class is the parent level class for all child level classes that generate random numbers from different distributions. Since all distributions generate their values by first getting a U(0,1) random number, Rand supplies this and then the child distribution class transforms this U(0,1) random value into the appropriate distribution. Rand is an abstract class, and therefore cannot be directly instantiated. The children of class Rand can all be instantiated. Rand provides high level data and behavior for the children classes. To generate a U(0,1) random number, Rand first generates a new seed value from the previous seed value with the following equation.

$$seed(i+1) = (multiplier * seed(i)) \bmod (2^{31} - 1) \tag{2}$$

To generate a U(0,1) number from the new seed value, we normalize the new seed with:

$$U(0,1) = seed(i+1) \div (2^{31} - 1) \tag{3}$$

The seed is any 32 bit integer value between 1 and 2^31-2 inclusive. Selection of the initial seed is quite arbitrary when using just one generator because every seed value starts a stream which will not repeat for more than two billion numbers. It is important that two different generators not start at the same seed value because then they would produce the same numbers possible resulting in some correlation of output in the simulation.

The multiplier could also be any 32 bit integer value between 1 and 2^31-2 but the generator will act poorly unless the multiplier is a primitive element modulo 2^31-1. Much research has been conducted to identify the many positive primitive roots of m = 2^31-1 (Fishman and Moore, 1985). Fishman and Moore identified 414 multipliers out of more than 534 million possible candidate multipliers which had good acceptable properties. More stringent tests reduced the recommend multipliers to a pool of 207 multipliers. Fishman and Moore recommend 5 of the 207 as having outstanding statistical properties. Figure 18 lists the 5 best multipliers from the PMMCG with m = 2^31 -1.

5.2.1 Eighteen different generator classes

The eighteen different generator classes listed in figure 18 were created so that the simulation engine would be capable of conducting stochastic simulations. The children random number generator (RNG) classes get a U(0,1) random value from their parent, and transform this into a value from their distributions.

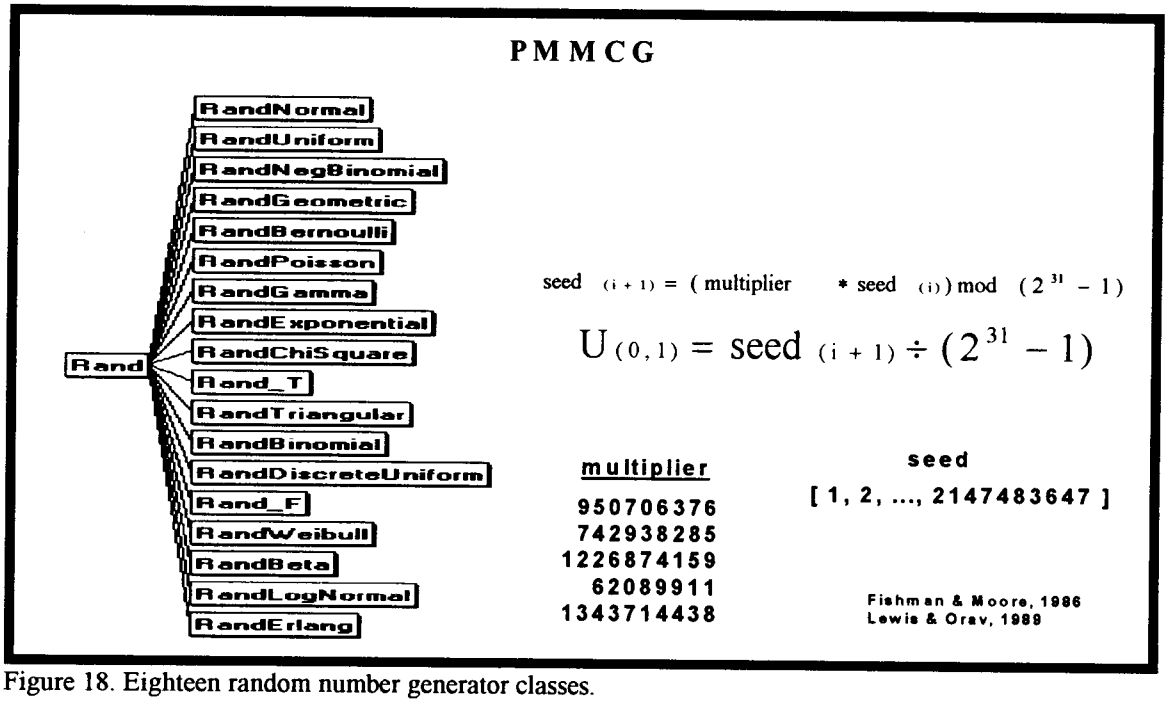


Figure 18. Eighteen random number generator classes.

One method of the random number object called RandValue, will return a random value from the appropriate distribution. RandValue is a pure virtual method and this is the only method that a user has to use to get random numbers. Since RandValue is a pure virtual method, any child object that is implemented must implement the RandValue method with the same arguments and return type.

SetSeed will set the seed for the next number generated. Any 32 bit integer between 1 and 2^31-2 is acceptable. Since there are over two billion values in this range, the random number generator will produce a stream of more than two billion unique random numbers before it repeats the sequence if an appropriate multiplier has been chosen.

SelectStream selects the multiplier for the PMMCG. There are about 414 acceptable multipliers but this class limits the built in selections to Fishman and Moore's 5 recommended multipliers and 3 more which have been used frequently in industry. Any of Fishman and Moore's 5 recommended multipliers could be used with confidence. Most simulation packages provide one good multiplier, but for comparison purposes, Rand will let the user select a multiplier.

5.3 The Stats abstract statistical collection class

Statistics are an important part of any simulation and although all statistics are collected by the simulation objects themselves, the ability to report them is handled through a simulation environment. SimEnv class provides basic statistic report capabilities and StatSimEnv provides replication and global statistic report features. If any certain type of statistics are collected for simulation objects in the simObjectList, then the simulation will automatically report them at the end of the simulation. Statistical flags like obsStats, timeStats, continuousStats, countStats, and distStats, inform the main program that headers for these statistics should be printed and then their statistics will be enumerated into reports.

The Stats class is an abstract class that provides statistical collection and calculation capabilities to other classes derived from it. Child level classes like StatsObs, StatsTime, and StatsContin can be instantiated to collect observational based, discrete-time based, and continuous-time based statistics. The child classes apply polymorphism to the collection and retrieval of statistics. Discussions on the collection of statistics during computer simulations can be found in Taha (1988).

The user can collect statistics on any simulation object by selecting a unique identification (ID) used in each type of statistic. Statistics gathering is therefore tied to each simulation object. An example of collecting continuous-time based statistics is given in figure 19.

```
Bog::Update( ... )
  {

  //-- update the object --//

  //-- update statistics --//
  AddContinObs( ID_PLANTS, plants, time );
  AddContinObs( ID_HERBIVORES, herbivores, time );
  AddContinObs( ID_CARNIVORES, carnivores, time );
  AddContinObs( ID_ORGANIC, organic, time );
  AddContinObs( ID_SOLAR, solar, time );

  Re-register object
  }
```

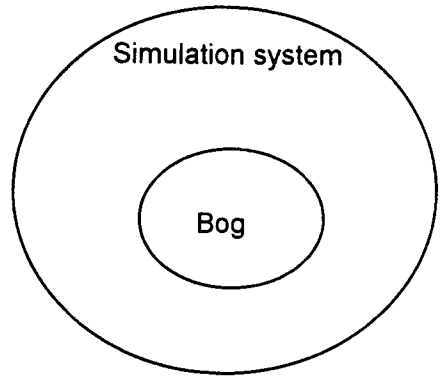


Figure 19. Collecting statistics in a simulation object.

5.3.1 Four collection classes

There are four different classes that collect statistics. These four statistical collection classes provide basic statistics for every type of stochastic simulation. The StatsObs, StatsTime, and StatsContinuous classes are children of class Stats. The Counter class is a stand alone class that provides counting statistics. The class hierarchy is given in figure 20, and computer source code is documented in Appendix I. The data and behavior of class Stats is inherited into the child classes by the hierarchical relationship. When the exact behavior of a child is not consistent with the parent class Stats, polymorphic virtual functions are used.

Class StatsObs collects observational based statistics for some unique ID. The class provides minimum, maximum, mean, variation, number of observations, and final value for the statistic.

Class StatsTime collects discrete-time based statistics for some unique ID. The class provides minimum, maximum, mean, variation, number of observations, and final value from some discretely changing variable.

Class StatsContinuous collects continuous-time based statistics for some unique ID. The class provides minimum, maximum, mean, variation, number of observations, and final value for some continuously changing variable.

The Counter class collects counting based statistics for some unique ID. This class provides a count, and number of observations for some variable.

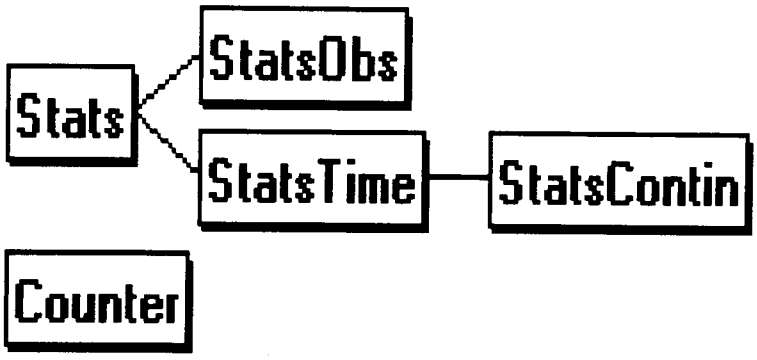


Figure 20. Class hierarchy for the statistical collection classes.

The statistical collection classes have much useful data. The ID for the statistic is an identification that is needed to differentiate between multiple similar statistics from the same object. The name of a statistic is a character string which will be output to reports. The value of a statistic is the current observation. Observations are recorded and temporarily stored in this data member. The minimum and maximum values are automatically updated during each observation of the statistic. The sum of the values is updated by summing the values and storing this information in Sx. The sum of the square values is updated by summing the square of the values and storing this information in SSx.

There are many useful public methods in the children classes. RecordValue records the current observation of the statistic. This method keeps the entire statistic up to date for each observation. GetName returns the name of the statistic. This name is used in automatic reports that list statistical output. GetMinimum returns the minimum of the statistic. GetMaximum returns the maximum of the statistic. GetMean returns the mean of the statistic. GetVariance returns the variance of the statistic. These methods are virtual at the parent because the children have different implementations. ReInitializeStats clears the current statistics. This method is invoked automatically by the simulation environment when the user wants to re-initialize statistics.

5.4 The ReplicationStats multiple run statistics class

When conducting stochastic simulations, the StatSimEnv class provides the automatic feature of replications through the Replicate method. Replicate controls the running of independent replications of a given experiment. It uses a callback function to a user defined experiment to run the replications. Everything about the system state, with the exception of the random number seeds, is automatically initialized to the same initial state for each replicate. Global statistics are automatically generated by the ReplicateStats type data member of the StatSimEnv. Computer code for the ReplicationStats multiple run statistics class is given in Appendix J.

The ReplicationStats class generates multiple run global statistics automatically from the statistics of each replication. The ReplicationStats class tracks replications with two data members, numReps and repNum. The variable numReps indicates the total number of replications to be run for the simulation. The variable repNum is the current replication number. Figure 21 shows the flow process when conducting replications.

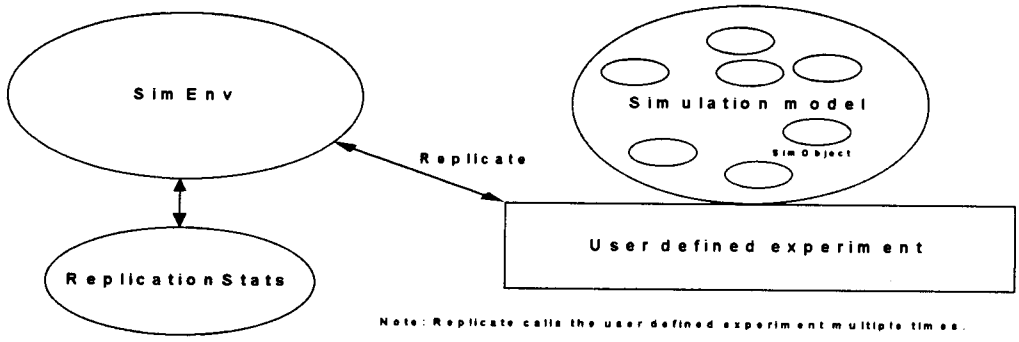


Figure 21. The global statistics class ReplicationStats.

There is a large amount of support data for the ReplicationStats class. The two main types of data are the tTable arrays and the statistical observation storage arrays. The tTable arrays are used in the construction of confidence intervals about some mean statistic. The statistical observation storage arrays hold certain statistics for each replication. The important statistics that must be saved during each replication are; means, minimums, maximums, and number of observations. This data is used at the end of a multi-run simulation experiment to calculate means, standard deviations, minimums, maximums, average number of observations, and confidence intervals.

ReplicationStats provides several public methods to control the behavior of the object. The class constructor ReplicationStats initializes the objects data and sets a reference to the simulation environment.

The recording of statistical information during each run is accomplished with the various Set methods. Global statistics are made available after invoking the method SetUpGlobalStatsOutput.

The output of global statistics is provided transparently by the StatSimEnv class. After the total replications have been run, the StatSimEnv will interact with the ReplicationStats class and the global output report methods to provide a global summary statistical report to the user.

5.5 Numerical integration

The Integrator class consists of several integration methods. During construction of the Integrator object, a particular type of integration method is chosen. The object will use this integration method during future request to integrate some differential equations. The selection of a integration method is user specified and should be based on precision and speed of computation. The default method Runge Kutta 4th order (RK4) provides sufficient accuracy for many engineering and scientific simulations. Runge-Kutta Feldberg provides a 5th order (RKF45) solution with error tolerance and time-step adjustment.

The class format is very simple, and its functionality could be packaged separately or within another object like SimEnv. The complexity of the class is in the writing of the integration methods. The interface to the integrator would be exactly the same for all integration methods.

The integration method of the Integrator class expects a reference to a State method from a simulation object. This State method will be called by the Integrator's integration method to integrate the differential equations of the simulation object.

The State method is a function written for an object which list the differential equations to be integrated. The State method is invoked as a callback function from the Integrator's integration method. The Update and State methods demonstrate the way integration is implemented. State variables are referenced and initialized in an objects Update method, passed to the Integrator, which in turn calls the State method to integrate over the time step, and returned to the Update method to be updated for the integration step. Two methods from a simulation object, Update and State, work with one method from the Integrator object to update the state of the object continuously over the time step. Examples of methods which include a call to the Integrator from the Update method for the RK4 and RKF45 are given in figures 22 and 23. Appendix K provides computer code for the Runge-Kutta integrators.

5.5.1 Runge-Kutta 4th order (RK4)

RK4 is the most popular Runge-Kutta integrator. It is practical for many engineer and science problems because it has error less than 10^(-m) where m is the order of integration and it is simple to program. (Yakowitz et. al., 1986, pp., 295-306)

RK4 uses a Taylor series approach to approximate a polynomial without taking derivitives. RK4 is considered the classic Runge-Kutta integration formula. Although RK4 is appropriate for most engineering problems, higher order Runge-Kutta integrators do enhanced the accuracy of the solution.

Runge-Kutta 4th order calls a State method, which holds the state variables and derivatives being integrated, 2 times during the integration. RK4 does not employ variable step size control. Variable step size control is a feature used to speed up simulations by integrating over some maximum step size if the integration error is within some user specified tolerance limits.

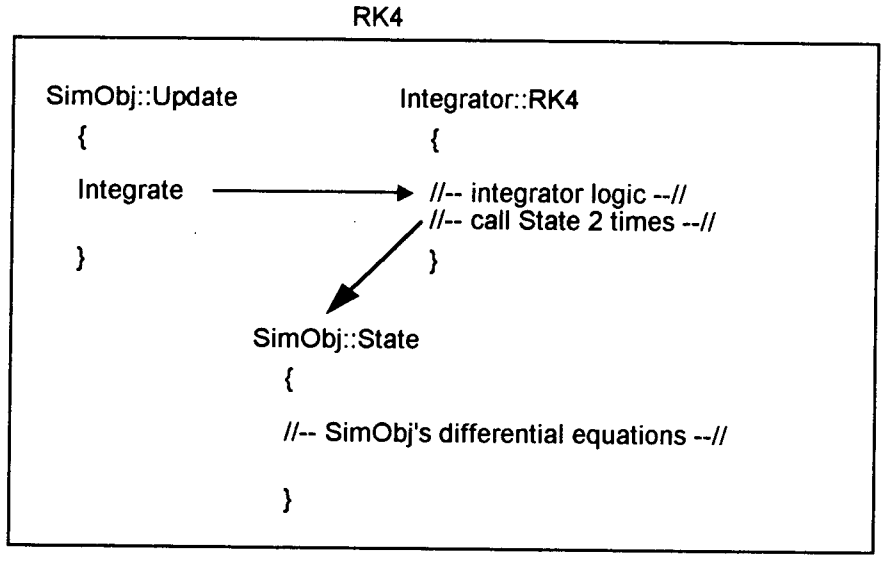


Figure 22. Simulating with 4th order Runge-Kutta.

5.5.2 Runge-Kutta Feldberg (RKF45)

RKF45 imbeds a 4th order Runge-Kutta Fehlberg step into a 5th order step solution. RKF45 (Fehlberg, 1969) implements a variable step size and absolute and relative error single-step integration solution. RKF45 calls the State method 3 times during the integration. The advantage of RKF45 over Runge-Kutta 4th order is that the step size can be adjusted on the fly. If we are not achieving the desired accuracy in our integration we can decrease the step size. Likewise, if we are achieving the desired accuracy in an integration, we can integrate at some maximum pre specified step size. Variable step size control in continuous simulations can result in faster, more precise simulations. Programs that implement variable step size control check the accuracy of the last integration step and adjust the step size accordingly. RKF45 is an integration procedure that implements variable step size control.

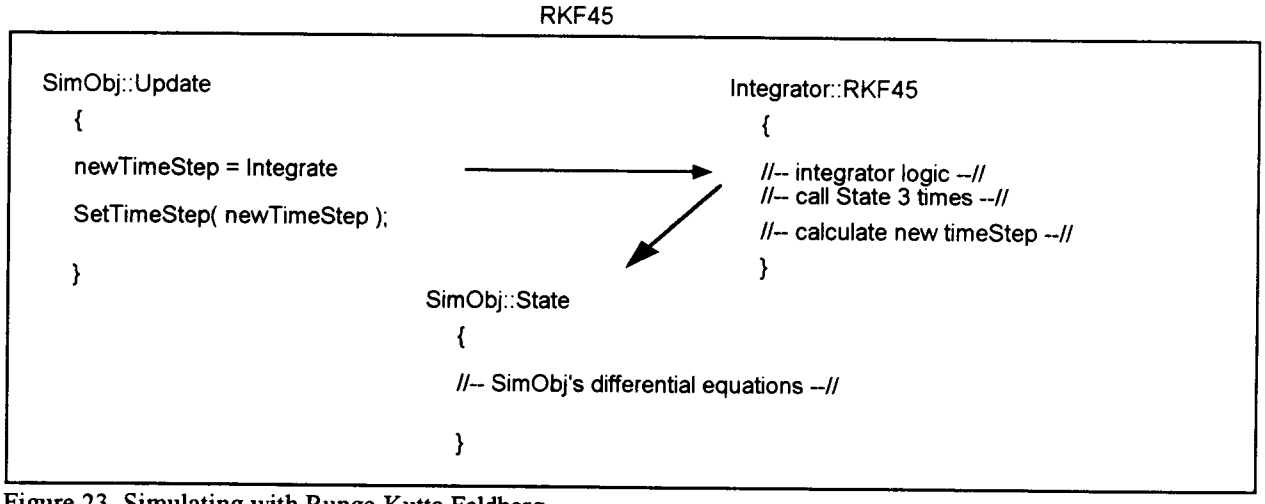


Figure 23. Simulating with Runge-Kutta Feldberg.

6. INTELLIGENT SIMULATION CAPABILITIES

Simulationists have long wanted to add AI capabilities into simulation languages. Unfortunately, many of the languages that were useful for implementing AI techniques were not appropriate for developing simulation languages. OOP languages overcome these previous problems and are very good for both AI and simulation. Adding intelligence to a simulation model is just a matter of developing intelligent simulation objects that inherit AI capabilities (Figure 24). These intelligent simulation objects can be called Agents or Experts. Agents are typically intelligent simulation objects that know how to answer questions and solve problems for other simulation objects. Experts are also intelligent simulation objects but they have added AI capabilities. The term Agent can be used interchangeable with Expert in the following discussion.

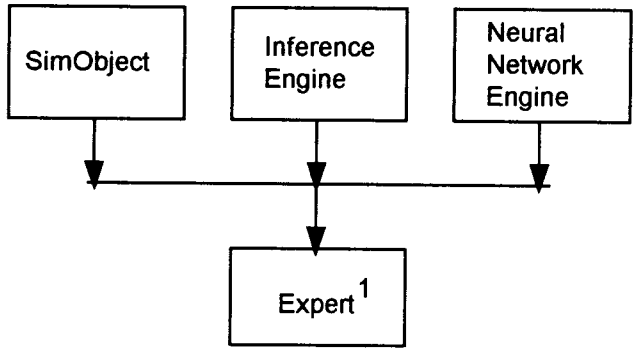


Figure 24. Intelligent simulation objects.

Experts assist other simulation objects during the running of a simulation model. Simulation objects can interact with Experts in four possible ways. The first three possibilities, Broadcasting messages, direct OOP messages, and Posting to a bulletin board, were discussed in chapter 4. The fourth approach is when an Expert observes the state of simulation objects from an external viewpoint. These different approaches to getting Experts involved in simulations allow for the most flexibility in bring AI into the simulation environment. If a real-time Expert is necessary, then the Broadcast or direct OOP message passing schemes should be used. If it is sufficient that actions are only taken during events, then the bulletin board or observing Expert is probably a sufficient solution. Figure 25 presents the different ways that Experts can participate in a simulation.

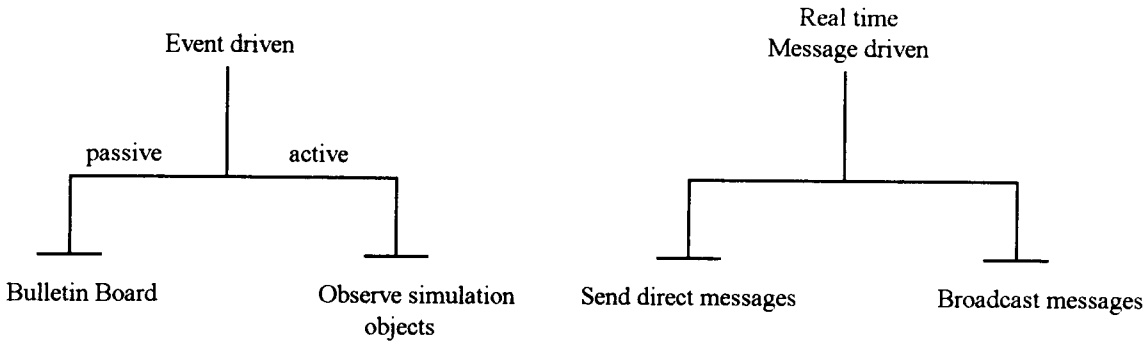


Figure 25. Four different ways that Experts can get involved in simulations.

6.1 Adding Expert's to the simulation

Since Expert's are just another simulation object, albeit a special simulation object, they are added to the simulation model just like any other simulation object. A high level view of a simulation system may look like figure 26.

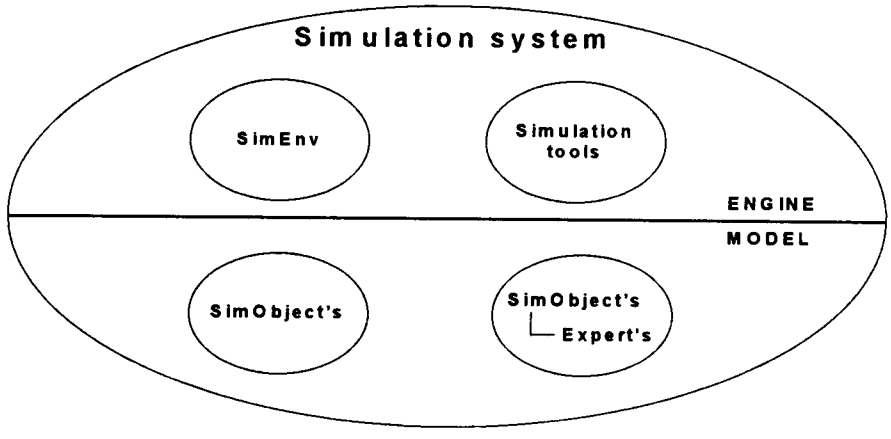


Figure 26. Components of a simulation system.

The process of adding Expert's to the simulation is very simple. It is exactly like adding any simulation object. If simulation objects want to participate in a simulation, then they must be registered with the simulation environment. The process of adding simulation objects to a simulation model was presented in section 3.2.

How Experts participate in a simulation is dependent on how quickly an Expert is required to solve some problem. Example 6.1 and 6.2 present different approaches to including Expert's into a simulation model.

Example 6.1. A military battlefield simulation is running and one of the simulation objects (M1 tank) observes another object in it's field of view. The M1 tank cannot identify the object so it requests expert assistance by Broadcasting a message to an Expert object (Commander) to identify the object. The Commander replies in real-time with a response.

Example 6.2. A salmon hatchery simulation is running and one of the simulation objects (fish lot) has reached a minimum oxygen threshold value for sustaining life. The fish lot takes action by Posting a message to the bulletin board where Fish Culturists Experts are known to check for problems. A fish Geneticist Expert simulation object is observing several fish lots that are approaching maturity. The fish Geneticist will take action as necessary.

In figure 25, the four different ways that Expert's get involved in simulations was presented. Example 6.1 is an example of the real-time message driven approach. Example 6.2 presented a scenario where the communication was event driven. The Fish Culturist Expert was considered a passive object since it checks the bulletin board during it's event cycle. The Geneticist Expert can be considered an active object since it is actively checking fish lots for problems.

One of the goals of this thesis was to provide a better or easier way to add intelligence to a simulation model. This chapter introduced the topic of adding intelligent simulation objects to a simulation model. In the next chapter, three examples from the literature verify the accuracy of the simulation engine. Section 7.2.1 shows how users of the simulation engine would add intelligent simulation objects into the simulation model.

7. SIMULATION ENGINE VERIFICATION AND DISCUSSION

In order to verify the accuracy of the simulation engine and demonstrate some of the advanced features of the OOP simulation tools, three examples are presented. The first two examples are of a continuous-deterministic simulation which was presented in Pegden et. al. (1990). Pegden provides continuous change output statistics for comparison. The third example is of a multi-server queuing system. Queuing theory is used to verify the accuracy of the output. All three examples conform very well to the standards. Further testing using textbook and theoretical solutions on different examples will further refine the simulation engine.

The advanced features used in the examples are inter-object communication and intelligent simulation capabilities. Detailed discussions on the general features of these topics were presented in chapters 4 and 6. In order to appreciate these added features, they are presented again but in the context of the examples. Finally, the difficulty in implementing them in a FORTRAN based simulation language like SIMAN is discussed.

7.1 Continuous-deterministic simulations -- The bog ecosystems

The bog ecosystems are useful for verifying the integration, statistical gathering, and inter-object communication capabilities of the simulation environment. Both systems verify the accuracy of the environment by comparing their results to the SIMAN based textbook solution (Pegden et. al., 1990, pg. 510). Figure 27 shows the two bog ecosystems. The first example presented in 7.1.1, is a one object system which incorporates five initial value differential equations. This example does not add anything new to the SIMAN implementation and it's purpose is just to compare the results. The second example presented in 7.1.2, is a multi-object system that uses an advance feature of the simulation engine called inter-object communication. Both OOP simulations use the RK4 integrator while the SIMAN package uses RKF45. Results are very similar and differences between the first example and the textbook solution should be attributed to the different integrators. The second example compares favorable to the first example and therefore lack of simultaneous solutions in the second example has no effect on the results. Perhaps this could be an area of concern if a larger time-step or a more difficult example had been presented.

Another factor to consider when implementing a single or multiple object system is the speed of computation. Both examples were timed and the multi-object system, which incorporates inter-object communication was found to be 2.4 times slower. This overhead is a result of the extra calls necessary during the integration. This extra simulation time must be taken into consideration when a user implements such models. Whether the benefits of such an approach outweigh the extra computation required is an issue that must be decided by the modeler. Further refinements to the simulation engine will likely decrease the CPU time involved in independent inter-object communication.

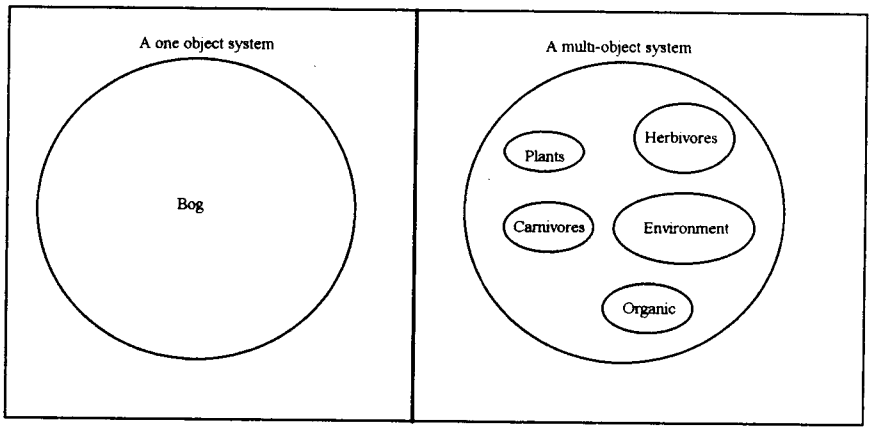


Figure 27. The bog ecosystems.

7.1.1 A one object system

The model is a continuous representation of the lake ecosystem and consists of three species, solar energy input, organic lake-bottom sediment formation, and loss to the environment. These six components are measured by energy content in calories per square centimeters. The three species -- plants, herbivores (fish that eat plants), carnivores (fish that eat fish) -- and organic sediment formation, and environment variables make up the bog ecosystem. The solar energy input is a non-controllable exogenous variable. The system is modeled using five initial value ordinary differential equations and one state equation for solar radiation (Table 5).

Table 5. Differential equations for the bog ecosystem.

```
solar = 95.9 * ( 1.0 + 0.635 * sin( 2.0 * PI * time );

plantRate = solar - 4.03 * plants;
herbivoreRate = 0.48 * plants - 17.87 * herbivores;
carnivoreRate = 4.85 * herbivores - 4.65 * carnivores;
organicRate = 2.55 * plants + 6.12 * herbivores + 1.95 * carnivores;
environmentRate = 1.0 * plants + 6.9 * herbivores + 2.7 * carnivores;
```

The bog ecosystem is presented for comparison purposes. The results of the OOP simulation output will be compared to the textbook solution given by Pegden.

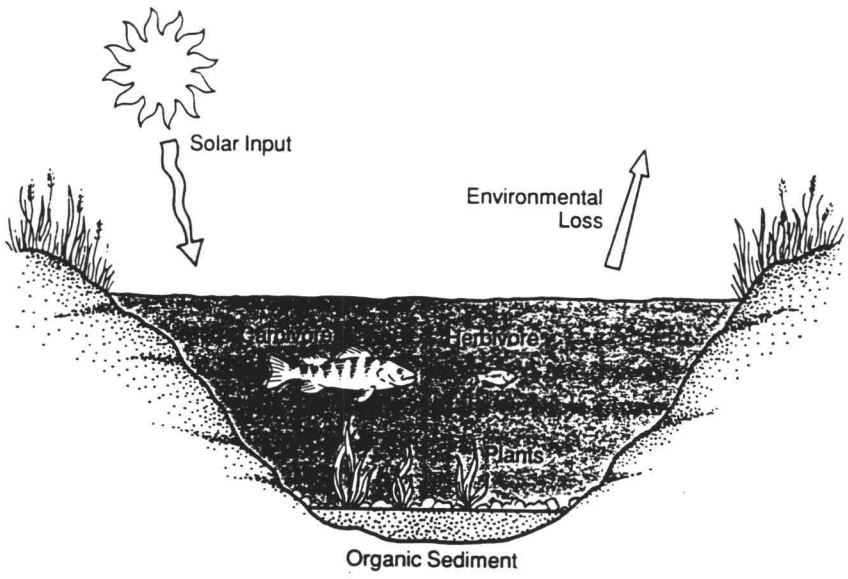


Figure 28. The cedar bog lake ecosystem. (Pegden et. al., 1990, pg. 507)

The bog ecosystem simulation will be described in a step by step procedure so that the reader may better understand the process the simulation environment goes through to simulate the bog simulation object.

Step 1: Register simulation objects.

Event is automatically placed on the event list.

Event list

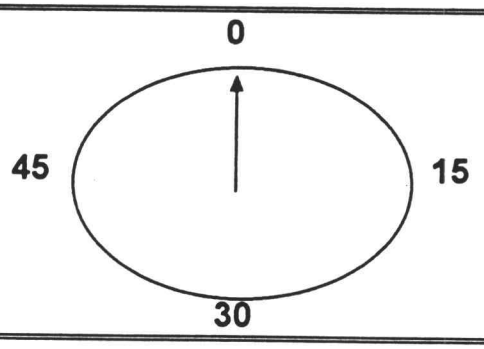
simulation object	event tick
pBog	0

Step 2: Start simulation clock.

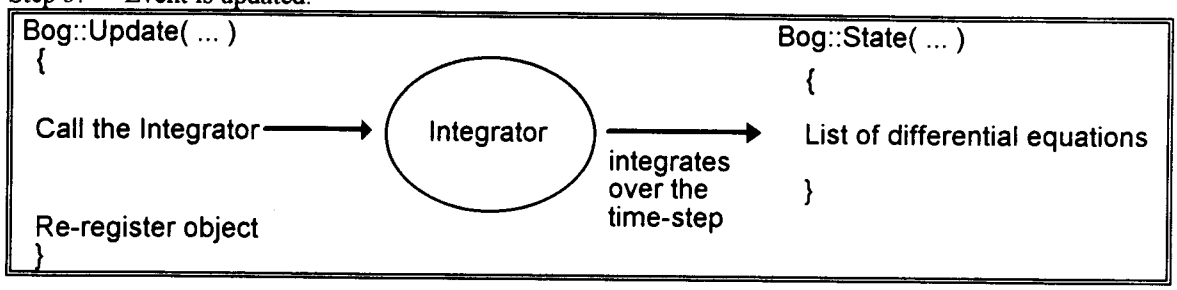
First event is removed from the event list.

Event list

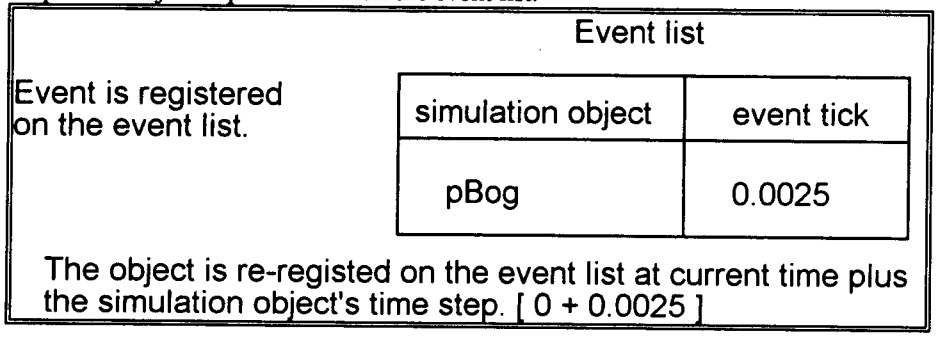
simulation object	event tick



Step 3: Event is updated.



Step 4: Object is placed back on the event list.



Step 5: Process repeats until the simulation stop time.

The process repeats with the event removed from the top of the event list, the simulation clock is updated to the events simulation time, and then the object is updated.

The five steps of the "one object system" were presented to demonstrate the general process of simulating a simulation object with the simulation engine. The simulation results for the single object system are in agreement with the textbook solution and therefore help to verify the accuracy of the simulation engine (Appendix E). Simulationist's would implement a one object system if minimum computation time or simultaneous solutions were an issue.

The implementation in a FORTRAN based simulation language such as SIMAN would be similar. The SIMAN program would package the differential equations in a function called STATE much like the Bog's 'State' method. There are no obvious advantages to implementing the single object OOP system versus a SIMAN based solution.

7.1.2 A multi-object system

The multi-object system which incorporates inter-object communication into the design of the system, results in a more realistic simulation. This type of design is where OOP really shines. Seperating objects into their natural components is advantageous because the objects have a direct association with there real life conterparts. There are several benefits listed in table 6, but the most obvious is increased modularity.

Table 6. Benefits of a multi-object system.

1. Good for building GUI's: one image is one object.
2. Makes code more reusable in other simulations.
3. Makes the system seem more realistic.
4. Differential equations can be defined through the GUI if one object has one equation.
5. The system is more modular.
6. A simulation object has it's own time-step.

Along with the benefits in any system, there is usually some disadvantages. The design of a multi-object system that separates the differential equations among the separate simulation objects has some disadvantages (Table 7)

Table 7. Disadvantages to a multi-object system.

1. Solutions to differential equations are not simultaneous.
2. Computation time is significantly lower.

The process involved in simulating the multi-object bog ecosystem that was presented in figure 27 will be described so the reader will better understand the inter-object communication capabilities. By utilizing the inter-object capabilities of the simulation environment, we can separate the simulation objects into their real life state. The bog ecosystem is not one object but multiple objects which have their true identity.

Step 1: Register simulation objects.

Event is automatically placed on the event list.

Event list

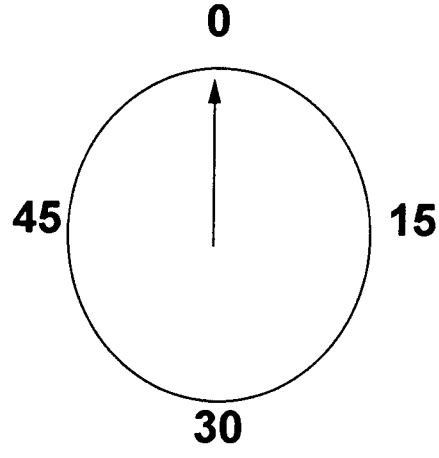
simulation object	event tick
pPlants	0
pHerbivores	0
pCarnivores	0
pOrganic	0
pEnvironment	0

Step 2: Start the simulation clock.

First event is removed from the event list.

Event list

simulation object	event tick
pHerbivores	0
pCarnivores	0
pOrganic	0
pEnvironment	0



Step 3: Event is updated.

The Plant object is updated over the time-step. This process of updating the object is similar to the updating of the bog object in the "one object system". After the Plant object is updated, a more interesting process occurs when the Herbivore object is updated because as indicated in table 5, which lists the differential equations, the differential equation for Herbivores requires the value of both Plants and Herbivores.

After the Plant object is updated over the time-step, it is re-registered onto the event list.

Step 4: Object is placed back on the event list.

Event is registed on the event list.

Event list

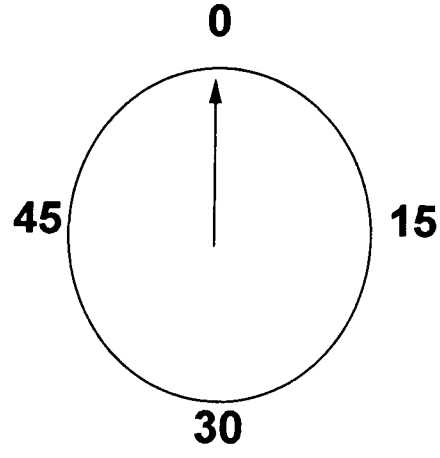
simulation object	event tick
pHerbivores	0
pCarnivores	0
pOrganic	0
pEnvironment	0
pPlants	0.0025

Step 5: Next event removed from event list.

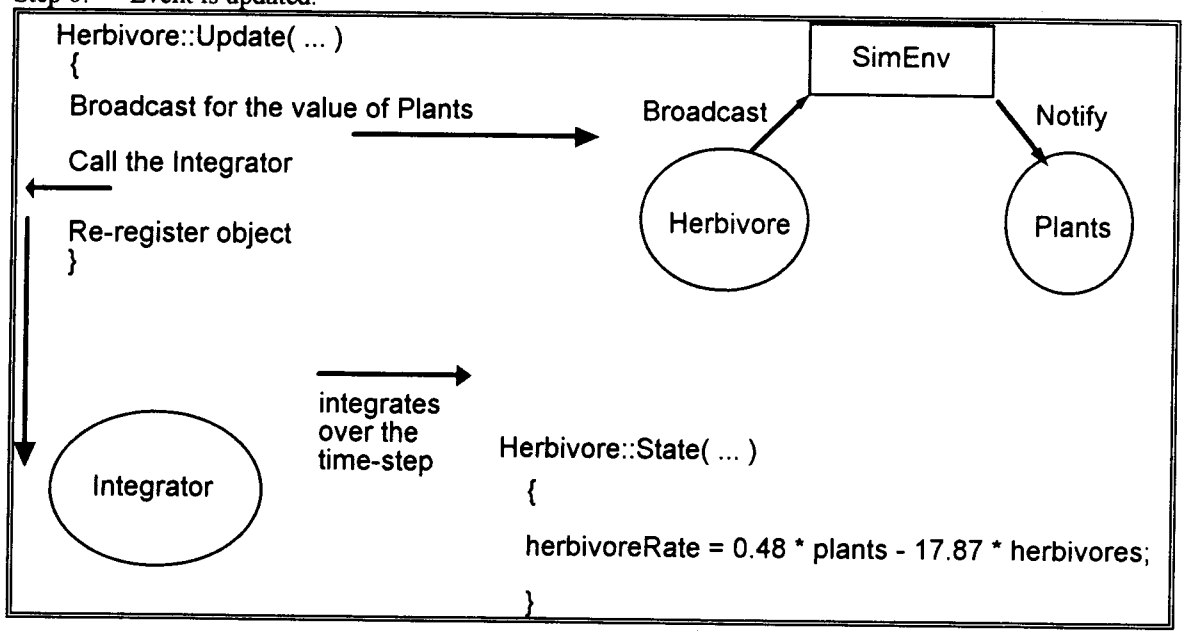
First event is removed from the event list.

Event list

simulation object	event tick
pCarnivores	0
pOrganic	0
pEnvironment	0
pPlants	0.0025



Step 6: Event is updated.



Step 7: Process repeats until the simulation stop time;

The process repeats with the top most event being removed from the event list, the simulation clock is updated to the events simulation time, and then the object is updated.

The seven steps of the "multi-object system" were presented to demonstrate how separate objects can communicate with object independent message passing to simulate a system through time.

The results of the multi-object bog ecosystem are presented in Appendix E. Output is comparable to the previous solutions and therefore helps in verifying the accuracy of the simulation engine.

If one wanted to implement the multi-object OOP example in a traditional simulation language like SIMAN, then we would have to make liberal use of global variables and function calls to get the state values of the other components in the system. The function names which stored the differential equations would need to be user defined and different. This would result in complex, confusing, unmanageble, and inefficient computer code. The multi-object OOP implementation is more elegant and understandable than any SIMAN model because of its modular packaging and virtually independent relationships among the objects. The OOP system uses the independent Broadcast method provided by the simulation environment to get the state values of the other components in the system. Implementing such an independent communication system in SIMAN would be very complex if not impossible.

7.2 Discrete-stochastic simulation -- A multi-server queuing system

A multi-server queuing system was implemented to demonstrate the discrete-stochastic simulation features of the simulation engine. It is also used to elaborate on the intelligent simulation and inter-object communication capabilities of the simulation engine as well as why this OOP approach is better than a SIMAN or FORTRAN based approach. First, the M/M/5 queuing system is presented and compared to theoretical solution, then a discussion on a real world variation to the model is presented. Finally, a comparison between the OOP intelligent simulation and a SIMAN FORTRAN simulation is outlined.

> "Consider a bank with five tellers and one queue, which opens its doors at 9 A.M., closes its doors at 5 P.M., but stays open until all customers in the bank at 5 P.M. have been served. Assume that customers arrive in accordance with a Poisson process at rate 1 per minute (i.e., IID exponential inter-arrival times with mean 1 minute), that service times are IID exponential random variables with mean 4 minutes, and that customers are served in a FIFO manner. ..." (Law & Kelton, 1991, pg. 524)

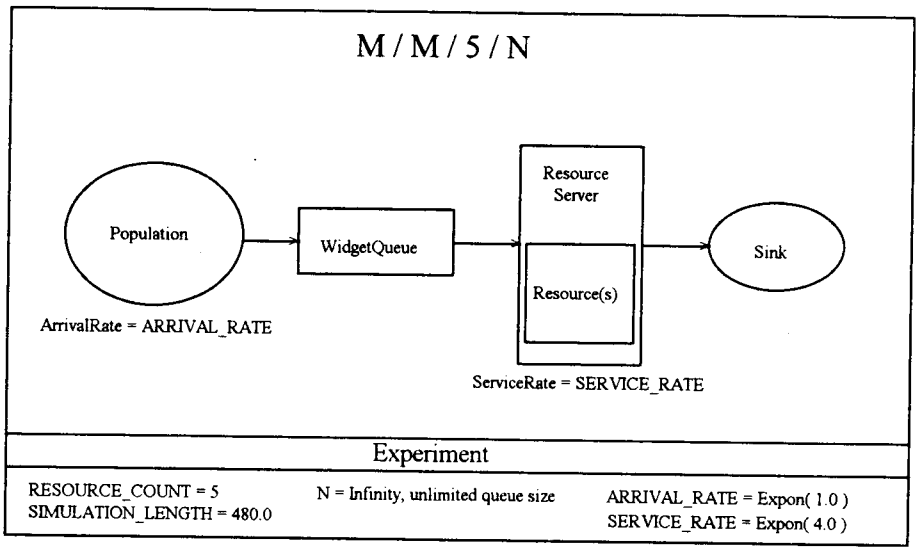


Figure 29. An M/M/5/infinity/FIFO steady state queuing model.

The model is a M/M/5/infinity/FIFO queuing model with an arrival rate of 1 customer per minute, and a service rate of 0.25 customers per minute. This is a multiple server, infinite waiting room scenario. This implies a rho = arrival rate / (num servers * service rate) = 0.80.

Queuing theory results for P0 and Lq were calculated using standard formulas (Ravindran, et. al., 1987, pg. 327). Then, all other queuing performance measures were calculated using standard equations for a steady state queuing model (Ravindran, et. al., pg. 314). The results are as follows:

Table 8. Theoretical results of the M/M/5/infinity/FIFO queuing model.

Parameter	Theoretical value	Measure
rho	0.8	utilization factor
P0	0.013	probability 0 customers in system
Lq	2.216	mean length of queue
L	6.218	mean length of system
B	4.002	mean number of busy servers
U	0.801	mean utilization
R	1	mean throughput rate
W	6.219	mean waiting time in system
Wq	2.216	mean waiting time in queue

The stochastic simulation model was run for 100 independent replications, assuming no customers are initially present at the beginning of each replication. The only items in the state of the system that differ at the beginning of each simulation run are the initial seed values for each random number distribution. The beginning seed values for runs 2 through 100 are the ending seed values for runs 1 through 99. Thus, the random number streams used are continuous from the initial seed values of replication number 1 through replication number 100. The output for 100 independent replications and the program code is presented in Appendix E. The results of the 100 independent replications are now compared to theoretical results.

Table 9. Theoretical vs. Experimental results for the M/M/5/infinity/FIFO queuing model.

Parameter	Theoretical value	Statistic	Experimental 95% Con-Limit Lower	Upper	#
Lq	2.2165	Len all	2.0564	2.6771	
B	4.0024	Busy	3.8969	3.9610	##
U	0.8005	ResServ	0.8218	0.8397	##
W	6.2188	Time sys	6.0709	6.6880	
Wq	2.2165	Time all	2.0530	2.6286	

\# not a Bonferroni confidence interval
results not within 95% confidence limits

The results of the discrete-stochastic simulation compare well with the theoretical results with the exception of the statistics collected by the resource server. The resource server collects statistics on the resources by observing their busy time and overall utilization. This area of statistic collection will be

improved in the future. The OOP simulation results are in agreement with the queuing theory solution and therefore help to verify the accuracy of the simulation engine.

It was very important that the simulation engine could correctly perform against a queing model which had an analytical solution. This example was a starting point to a more realistic simulation model. Rarely do we find such pure examples in real life. The simulation engine was designed to be flexible and allow for the inclusion of real life experts into a simulation model. This is one of the major differences between this simulation engine and other simulation languages.

7.2.1 Adding an Expert to the queuing system

Imagine the complexity added when the M/M/5 queuing model becomes more real world and Experts are required in the system. Although this example is a simple one, the reader is encouraged to think about other situations where such additions may be necessary.

> "Consider the same bank presented in section 7.2 with the addition of an Expert. Experts must approve special transactions that the tellers conduct. Any supervisor and several well qualified employees of the bank are capable of approving such transactions."

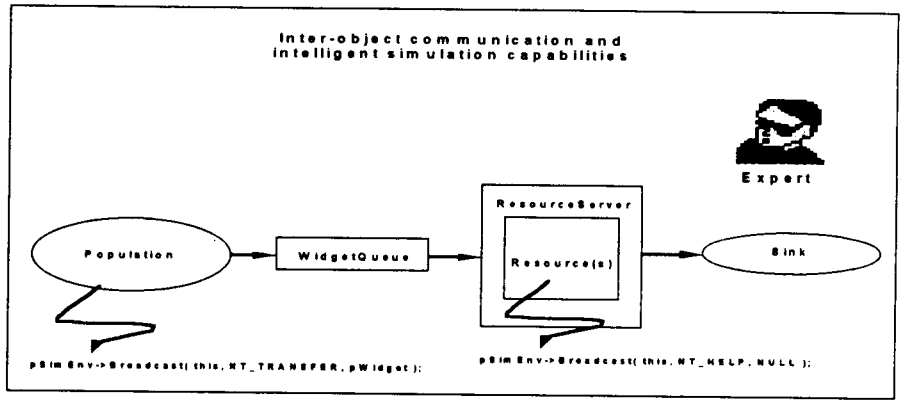
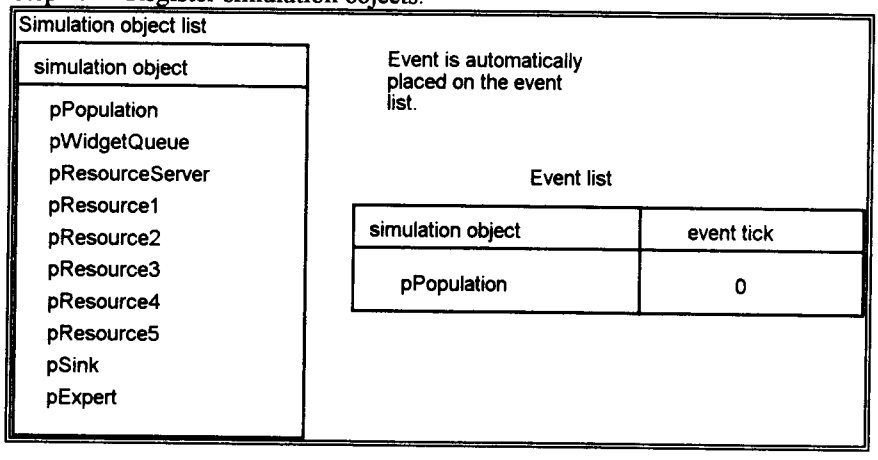


Figure 30. An M/M/5/infinity/FIFO steady state OOP queuing model with an Expert.

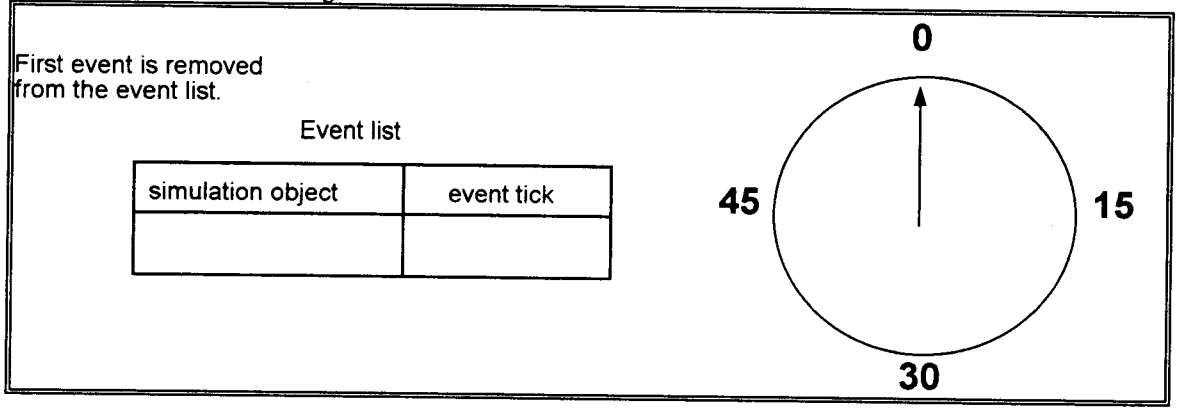
The different approaches to implementing the addition of an Expert to the simulation model of the OOP simulation engine and a SIMAN FORTRAN based implementation should help the reader in understanding why the OOP approach is better. Also, the fact that there are no known textbook or journal examples in SIMAN, SLAM, GPSS, or FORTRAN which integrate AI into a simulation model should further encourage the reader into looking toward a OOP based simulation environment. The simulation engine provides advanced features which allow the easy integration of AI objects into the simulation model.

In figure 30, the Population object Broadcast's to the system that it wants to transfer a Widget into the system. Any object that has added a notification filter to the simulation environment can have the opportunity to receive this Widget. The transfer of a Widget is extremely simple from the users viewpoint, and does not even require that the receiving object be part of the flow process. Also, consider the addition of Experts into the system. When a Resource needs Expert assistance, one simple call to the system (pSimEnv->Broadcast(this, NT_HELP, NULL)), fulfills this need. To help the reader better understand the process that the simulation environment goes through to send a message to an expert, a list of steps in the M/M/5 queing system follows. Figure 30 will help the reader understand the system and flow process.

Step 1: Register simulation objects.

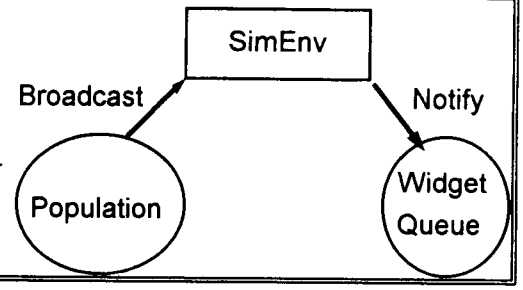


Step 2: The simulation begins with the first event.

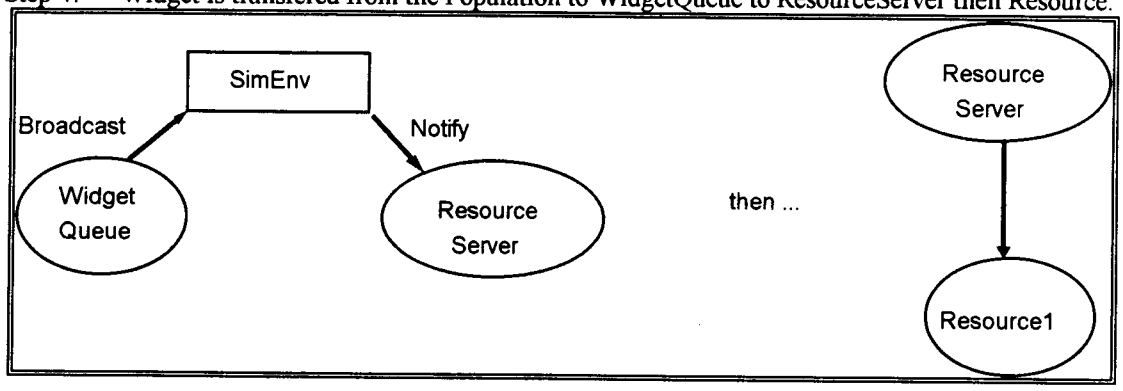


Step 3: The event is updated.

```
Population::Update( ... )
 {
 timeStep = Random value
 Create a Widget object
 Broadcast a transfer to the system
 Re-register object
 }
```



Step 4: Widget is transfered from the Population to WidgetQueue to ResourceServer then Resource.



Step 5: When the Widget reaches Resource1, it registers the departure event for the Widget.

Event list		
simulation object	event tick	event
pPopulation	60	ET_GENERATE
pResource1	240	ET_DEPART

Step 6: The process continues.

The process continues with Widgets being added to the system and Resource(s) serving them. Events are registered onto the event list as time progresses forward. Now consider when one of the resources needs expert assistance.

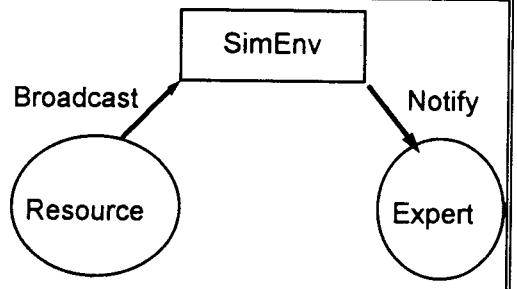
Step 7: One of the Resource(s) calls for help.

```
Resource::Update( ... )
 {
 timeStep = Random value

 Broadcast for help to the system ------>

 Register the departure time for the Widget

 Re-register object
 }
```



Note: One Broadcast from Resource could result in many Notifications to Experts.

This simple approach provided by the simulation engine would not be so simple in a SIMAN implementation. To implement such a system in SIMAN, a programmer would have to set up multiple function calls in a nested if statement, which would introduce code dependencies and make the model less flexible and more complex. Contrary to the OOP simulation environment which provides a technique for independent communication and the integration of AI into the simulation model, the SIMAN and FORTRAN languages are clearly deficient of such capabilities. The Expert in this example dosen't look like it adds much to this example. On the contrary, this example was given to show that there is nothing different between adding simulation objects and adding Expert simulation objects. Expert and AI type simulation objects are very similar to simulation objects except for the fact that they inherit AI capabilities.

It is probable that the use of AI in simulation modeling will only increase in the future. Languages and tools that provide such capabilities will undoubtedly become increasingly popular.

8. CONCLUSIONS

The necessity of computer simulation in the analysis of complex systems will play an ever increasing role in inquiry and decision making. Simulation analysis provides a means which an analyst can use mathematical and statistical information to assist in decision making of real world systems. As the cost of computers continue to drop and power continues to increase, CPU time will have less importance than analyst time. Decreasing the amount of time it takes to develop simulation experiments is an important topic for further research. Development of intelligent user interfaces which assist in the task of experimental design, statistical analysis, and model building will have a significant impact on the increasing popularity of computer simulation.

The generally high cost of automated systems will require corporations to simulate a system before implementation. One government agency, The US AirForce, is requiring simulations on many proposed automated systems. Experimental design and statistical analysis, an important element in simulation modeling, would be more readily used if users were provided an effective interface to conduct simulation experiments. GUI's hold great promise in reducing the complexities of creating simulation models.

Good software tools are necessary for the future development of simulations and simulation languages. Object oriented programming languages are a perfect accompaniment for the creation of GUI's and simulation tools. Experts and simulation activists need to continue the development of reusable simulation tools.

This thesis has given the author the opportunity to explore the theory and implementation of both traditional and novel simulation tools. Several traditional simulation tools like; statistics gathering, random number generators, integrators, debugging, and reporting features were created to provide basic simulation requirements. Some novel concepts like inter-object communication and the inclusion of intelligent simulation objects into a simulation model were developed to fulfill a need to create intelligent and more real world simulation models. The communication methodologies Broadcast-Notify and Broadcast-Notify-Transfer are novel ideas never seen in other simulation languages.

The objectives of this thesis were to layout the basic components of a modern commercial grade simulation language. These objectives have been met. The author, colleagues, and readers of this thesis should be able to take the ideas, source code, and simulation examples provided and begin development of a commercial grade simulation language.

This work is not without its limitations. As was mentioned in the chapter 1 (INTRODUCTION), the three main components of a commercial grade simulation language would be a simulation engine, a simulation model framework, and standard packaging for simulation experiments. The experiment packaging was not completed due to time and resource constraints. Presently, models can be developed with default experiments but no standard interface is provided for the running of different experiments.

Future enhancements to this simulation language would provide a standard interface for the running of simulation experiments. Perhaps an experimental design module would create interface files that could be run by the simulation engine. Developments in data base technology is accelerating. Standard data base hooks into the simulation engine would be useful.

Simulation is a very popular analysis tool for scientists and engineers. It is also considered one of the ten most critical technologies by the Department of Defense. The development of this simulation language and others like it will provide important tools necessary for the analysis of complex systems.

BIBLIOGRAPHY

APL, 1970, APL/360 with Statistical Applications, Addison-Wesley Publishing Co.

Ashayeri, J., Gelders, L., "Production planning evaluation of PCB assembly through simulation", Proc. 4th Int. Conf. Simulation in Manufacturing, Nov 1988, pp. 63-70.

Banks, J., Carson, J. S., 1984, Discrete-event system simulation.

Belanger, R., Donovan, B., Morse, K., and Rockower, D., 1990, MODSIM II Reference Manual - Revision 6, CACI Products Company, La Jolla, CA.

Bischak, D. P., Roberts, S. D., Object Oriented Simulation, Proceedings of the 1991 Winter Simulation Conference.

Blake, K., and Gordon, G., Systems Simulation with Digital Computers, IBM Systems Journal, III, No. 1, 1964.

Borland C++ Programmers Guide, 1992, Version 3.1, Borland International.

Bulgren, W.G., 1982, Discrete System Simulation, Prentice Hall.

Charniak, E., and McDermott, Drew, 1985, Introduction to Artificial Intelligence, Addison-Wesley Publishing Company, Inc.

Cochran, W. G., and Cox, G. M., 1957, Experimental Designs, 2nd ed., Wiley.

Cox, B., 1986, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, MA.

Dahl, O. J., Nygaard, K., SIMULA -- An Algol-Based Simulation Language, CACM, vol. 9, no. 9, 1966, pp. 671-687.

Dahl, O. J., Nygaard, K., SIMULA, Introduction and User's Manual., Norwegian Computing Center, Oslo, 1967.

Dahl, O. J. and Nygaard, K., 1967, Simula: A Language For Programming And Description of Discrete Event Systems", Fifth Edition, Norwegian Computing Center, Oslo.

Diamond, W. J., 1989, Practical Experiment Designs for Engineers and Scientist, 2nd edition, Van Nostrand Reinhold.

Emshoff, J. R. and Sisson, R. L., 1970, Design and use of Computer Simulation Models, The MacMillian Company.

Faber, M., Lipper, E.H., "A study of load balancing algorithms for Just-in-Time operations", Proc. 4th Int. Conf. Simulation in Manufacturing, Nov 1988, pp. 33-43.

Fehlberg, E., Low-Order Classical Runge-Kutta Formulas with Step-Size Control and Their Application to some Heat Transfer Problems, NASA report TR R-315, 1969, Huntsville, Alabama.

Fehlberg, E., New-High Order Runge-Kutta Formulas with Stepsize Controls for Systems of First and Second Order Differential Equations, Z. Angew. Math. Mech., 44, 1964, pp., 83-88.

Fisher, J. A., 1992, Unpublished Object Oriented Simulation classes, Oregon State University.

Fisher, J. A., and Bolte, J. P., 1992, Unpublished Object Oriented Simulation Engine, Oregon State University.

Fisher, R.A., 1935, "Design of Experiments," (Oliver & Boyd, Edinburg).

FishMan, G. S. Concepts and Methods in Discrete Event Digital Simulation, 1973, John Wiley & Sons, Inc.

Fishman, G.S. 1978, "Principals of Discrete Event Simulation", (Wiley: New York).

Fishman, G. S., and Moore, III, L. S. "An exhaustive analysis of multiplicative congruential random number generators with modulus 2^31 - 1", SIAM J. Sci. Stat. Comput, Vol. 7, No. 1, January 1986, pgs., 24 - 45.

Fishman, G. S., and Moore, L. R., 1982, A statistical evaluation of multiplicative congruential random number generators with modulus 2^31-1, J. Amer. Statist Assoc., 77, pp. 129-136.

Fishman, G.S., 1973, Concepts and Methods in Discrete Digital Simulation, John Wiley & Sons.

Forsythe, G., Malcolm, M., and Moler, C., Computer Methods for Mathematical Computations, 1977, Prentice-Hall, Inc.

Franta, W. R., 1977, The Process View of Simulation, North-Holland Publishing Co.

Gordon, G., 1969, System Simulation, Prentice-Hall, Englewood Cliffs, N.J.

Hastings, N.A.J., and Peacock, J.B., 1975, Statistical Distributions, Halsted Press.

Heiberger, R. M., 1989, Computation for the Analysis of Designed Experiments, John Wiley & Sons, Inc.

Helsgaun, K., 1980, DISCO -- A SIMULA-based Language for Continuous, Combined, and Discrete Simulation, SIMULATION, July, pp. 1-12.

Hoover, S.V., and Perry, R.F., 1989, Simulation, Addison-Wesley Publishing Co.

IBM, 1970, General Purpose Simulation System/360 User's Manual, GH 20-0326, White Plains, N.Y.

IMSL, 1987, IMSL Stat/Library. IMSL Stationary: Users Manual, Version 1.0. IMSL, Inc., Houston

Kelton, W. D., 1987 Winter Simulation Conference proceedings : December 14-16, 1987.

Kivat, P. J., Villanueva, R., Markowitz, H. M., The Simscript II programming language, [1969, c1968], Prentice-Hall series in automatic computation.

Kiviat, P.J., Digital Computer Simulation: Modeling Concepts, The Rand Corporation, RM-5378-PR, Santa Monica, CA., 1967.

Knuth, D. E., 1981, "The art of computer programming Vol 2: Semi-numerical Algorithms", 2nd ed., Addison-Wesley, Reading, MA.

Law, A.M., and Kelton, W.D., 1991, Simulation Modeling & Analysis, McGraw-Hill, Inc.

Ledbetter, W.N., Cox, J.F., "Are OR Techniques being used", Industrial Engineering, pp. 19-21, Feb 1977.

McHaney, R., 1991, Computer Simulation: A Pratical Perspective, Academic Press, Inc., pp 110-111.

McLeod, J., 1982, Computer Modeling and Simulation: Principals of Good Pratice 10(2). Simulation Concils, Inc., La Jolla, CA.

Meyer, B., 1987, Reusability: The Case for Object-Oriented Design, IEEE Software, 4, 2, pg. 50-64.

Modern Materials Handling, Simulation in Manufacturing, Cahners Publication, Sept., 1987.

Montgomery, D. C., 1991, Design and Analysis of Experiments, 3rd ed., John Wiley & Sons, Inc.
N. Wirth. Algorithms + Data Structures = Programs. Prentice-Hall series in Automatic Computation, Englewood Cliffs, New Jersey, 1976.

Naylor, T. H., Balintfy, J. L., Burdick, D. S., and Chu, K., 1966, Computer Simulation Techniques, John Wiley & Sons.

Naylor, T.H., Balintfy, J.L., Burdick, D.S., and Chu, K., 1966, Computer Simulation Techniques. John Wiley & Sons, Inc.

Patil, G.P., Boswell, M.T., Joshi, S.W., and Ratnaparkhi, M.V., 1984, Dictionary and Classified Bibliography of Statistical Distributions in Scientific Work, Vol 1: Discrete Models, International Co-operative Publishing House.

Patil, G.P., Boswell, M.T., and Ratnaparkhi, M.V., 1984, Dictionary and Classified Bibliography of Statistical Distributions in Scientific Work, Vol 2: Continuous Models, International Co-operative Publishing House.

Pegden, D. C., Shannon, R. E., and Sadowski, R. P., 1990, Introduction to Simulation Using SIMAN, McGraw-Hill, Inc.

Petersen, R. G., 1985, Design and Analysis of Experiments, Marcel Dekker, Inc.

Press, W. H., Flannery, B.P., Teukolsky, S. A., and Vetterling, W. T., Numerical Recipes in C, 1988, Cambridge University Press. Chapter 15, 16.

Pritsker, A. A. B., 1979, Modeling and Analysis using Q-GERT networks, Halsted Press.

Pritsker, A. A. B., 1974, "The GASP IV Simulation Language", A Wiley-Interscience Publication.

Pritsker, A. A. B., 1979, Pegden, Dennis., "Introduction to simulation and SLAM", Halsted Press.

Pritsker, A. A. B., 1990, Papers Experiences Perspectives, Systems Publishing Corporation.

Pritsker, A. A. B., Kiviat, P. J., Simulation with GASP-II; a Fortran based simulation language, 1969, Englewood Cliffs, N.J., Prentice-Hall.

Ravindran, A., Phillips, D. T., and Solberg, J. J., 1987, Operations Research: Principals and Practice, John Wiley & Sons.

SAS, 1982, SAS User's Guide. SAS Institute, Inc., Cary, NC.

Schriber, T., 1972, A GPSS Primer, Ulrich's Books, Inc.

Schriber, T. J., 1990, An introduction to Simulation Using GPSS/H, John Wiley, New York.

Shannon, R. E., 1980, Engineering Management, Wiley.

Shewell, Steven.F., Buzacott, John.A., "Simulation and analysis of a circuit board manufacturing facility", Proceedings of the 1986 Winter Simulation Conference, pp. 686-693.

SIMAN(1990) Systems Modeling Corporation, Sewickley, PA.

Simscript II.5, CACI Products Company User Manual, 1988.

SimScript II.5(Kiviat, Villanueva, and Markowitz, 1983) CACI, SIMSCRIPT II.5 Programming Language, CACI Products Company, La Jolla, CA.

SLAM, 1989, Pritsker & Associates, Inc. West Lafayette, IN.

SPSS, 1983, SPSSx user's guide, McGraw Hill.

Stafford , Edward.F. Jr., Schroer, Bernard.J., "Simulation of a finished goods allocation system", Proceedings of the 1988 Winter Simulation Conference, pp. 652-661.

Strachey, C., Fundamental Concepts in Programming Languages. In Lecture Notes for the International Summer School in Computer Programming, Copenhagen, Denmark, 1967.

Stroustrup, B., Adding Classes to C: An Excercise in Language Evolution. Software Practice & Experiences, 13, 1983, pp. 139-161.

Systems Modeling Co., Sewickley, PA., 1990.

Taha, H. A., 1988, Simulation Modeling and Simnet, Prentice-Hall, Inc.

Thesen, A., and Travis, L.E., 1992, Simulation for Decision Making, West Publishing Co.

Websters New Collegiate Dictionary, 1975, C. & G. Meggiam Co.

Williams, R. B., Computer Simulation of Energy Flow in Cedar Bog Lake, Minnesota Based on Classical Studies by Lindeman, System Analysis and Simulation in Ecology, 1971, B.C. Patten, Editor, Academic Press.

Yakowitz, S., and Szidarovszky, F., An Introduction to Numerical Computation, 1986, Macmillan Publishing Company.

APPENDICES

Appendix A: Simulation terminology

attributes -- characteristics or properties of some object or entity.

checkpoint -- the process of saving the state of a simulation, usually in one or more data files, such that the state of the simulation may be reloaded and the simulation resumed at a future date. (See restart)

closed system -- a system with no exogenous variables.

continuous simulation -- a form of simulation in which models undergo changes strictly as a function of time. The model of an missile, for example, simulating the effects of lift, thrust, and drag based on differential or difference equations (where time is an independent variable) would be a continuous simulation. Computer simulations, generally, may be divided into continuous simulations and discrete-event simulations. (See discrete-event simulation)

continuous systems -- include real value variables that are characterized by smooth changes in state. (See discrete systems)

deterministic simulation -- a simulation for which a model's outputs are a function of the model's state and inputs. Random or pseudo-random number generators are not used. (See stochastic simulation)

deterministic system -- a system whose states are completely determined by it's initial state.

discrete systems -- include real value variables which assume particular values from a finite set of alternatives that are characterized by discontinuous changes in system state.

discrete-event simulation -- a form of simulation in which models undergo discrete state changes upon the detection of discrete-events. Discrete-event simulations, for example, are commonly used to simulate a network of computers which communicate via message passing. Computer simulations, generally may be divided into discrete-event simulations or continuous simulations. (See continuous simulation)

endogenous -- within the system

entity -- see object.

event driven simulation -- a simulation in which the passage of simulation time is determined from the next activity or event in time. The passage of time in event-driven simulations may occur in discrete jumps. (See time-driven simulation)

event or discrete-event -- a change which occurs at a point in time. Events often mark the beginning or ending times of activities, the occurrence of an action, or the passage of time. An event, for example, may indicate that an entity has begun to move, a message has been issued, or a time-out has occurred.

exogenous -- things that come into the system from the outside.

experimental design -- a set of rules by which treatments to be used in an experiment are assigned to the experimental units.

model or computer simulation model -- a set of instructions, often in the form of a programming language, which describe the operation of one or more physical entities. A model is typically capable of detecting changes in input conditions, representing internal activities, and issuing changes to output conditions. Models may be interconnected to represent a physical system.

open system -- a system with exogenous variables.

replication -- when treatments (same settings of parameters) appear more that once in an experiment, the treatment is said to be replicated.

restart -- the process of restoring the state of a simulator, usually from one or more data files, for the purpose of resuming a simulation run. (See checkpoint)

simulation clock -- represents the current simulation time at any point during a simulation. (See simulation time)

simulation or computer simulation -- the execution of a computer simulation model on a simulator. (See simulator)

simulation time -- time kept by a simulator for the purpose of synchronizing the computer simulation models. (See simulation clock)

simulator -- a customized computer program of a specific class of simulation model designed model the real world object.

state of the system -- a description of all objects, attributes, and activities and or events as they exist at some point in time.

stochastic system -- a systems whose state may take on different values given an initial state.

stochastic simulation -- a simulation in which random, or pseudo-random, numbers are used to determine timing and activity within a model. (See deterministic simulation)

system boundary -- the point in space that separates the endogenous from the exogenous variables.

system environment -- the objects surrounding the system.

time driven simulation -- a simulation in which the passage of simulation time is synchronized with an external time source. (See event-driven simulation)

time-step -- the time between simulation updates.

time unit granularity (TUG) -- the smallest unit of time which is recognized in a simulation, e.g., 1 second.

validation -- proof that a model is a correct representation of the real system.

verification -- proof that the simulation program is a realistic representation of the system being modeled.

Appendix B: Simulation constructs

Random number generation

Prime Modulus Multiplicative Congruential Generators (PMMCG) are the most popular and best studied generators available today. The form of the equation is;

$$X_i = (a^* X_{i-1}) \bmod m,$$

where m is prime, and a is a primitive element modulo m. If a and m are chosen such that the smallest integer b for which a^b - 1 is divisible by m is b = m - 1 (Knuth 1981, p., 19), then we can obtain every integer from 1, 2, ..., m-1 exactly once in each cycle. Thus (Xi-1) can be any integer from 1 through m-1 and a full period of m-1 will result. Using m = 2^31 -1 (2,147,483,647) and a as a primitive element modulo m, will produce the maximum cycle length possible with 32 bit integer arithmetic. Initial seed values can be any integer value from 1 to 2^31-2. A value of 0 or 2^31-1 will produce an infinite sequence of zeros. Once the generator is started with some initial integer seed value, it will eventually return every integer value between 1 and 2^31-2 exactly once before the cycle repeats. This will allow for more than 2 billion unique random numbers for each stream.

Fishman and Moore have proposed 5 optimal full period multipliers for the PMMCG with prime modulus 2^31-1. They present a battery of statistical tests for evidence of their statistical robustness (Fishman and Moore, 1982). They also discuss poor results of some previously used multipliers (Fishman and Moore, 1986). Some of the most popular multipliers are as follows:

Fishman and Moore (1986) -

Fishman and Moore recommend the best multipliers, all of which are positive primitive roots of m = 2^31-1. These recommendations are the result of an exhaustive computer tests on 267 million possible multipliers that allow maximum cycle length for PMMCG with modulus 2^31-1. The best values for a are:

a = 950,706,376
a = 742,938,285
a = 1,226,874,159
a = 62,089,911
a = 1,343,714,438

IMSL (1987) -

The IMSL Library provides a choice of three multipliers to be used with the PMMCG with modulus 2^31-1. Fishman and Moore (1986) reported poor performance for 397,204,094 and 16,807.

IMSL3 (1987) a = 950,706,376
IMSL2 (1984) a = 397,204,094
IMSL1 (1984) a = 16,807

SimScript II.5 (Kiviat, Villanueva, and Markowitz, 1983) -

SimScript uses one multiplier with the PMMCG with modulus 2^31-1. It has been shown to perform well by Fishman and Moore (1986), but does have some statistical weaknesses in three and six dimensions. The value is:

a = 630,360,016

SIMAN (1990) & SLAM (1989) -

These two simulation languages have much in common, including their generation of random numbers. One multiplier is used with the PMMCG with modulus 2^31-1. This multiplier is easily portable since it is small but it has been shown by Fishman and Moore (1986) to have deficient statistical properties unless a proper shuffling scheme has been implemented. The value is:

a = 16,807

SPSS (1983) & APL (1970) -

One multiplier is used with the PMMCG with modulus 2^31-1. This multiplier is easily portable since it is small but it has been shown by Fishman and Moore (1986) to have deficient statistical properties unless a proper shuffling scheme has been implemented. The value is:

a = 16,807

SAS (1982) -

SAS uses the PMMCG with two different multipliers. Fishman & Moore report good statistical results for both implementations. They are as follows:

a = 397,204,094
a = 16,807 with shuffling

The user can select up to 2^31-2 (2,147,483,646) different starting seed values. It is recommend that the selection of starting seed values be sufficiently spaced to prevent overlapping of random number streams. If only 10 unique streams were needed for a certain multiplier, the user could space the starting seeds by 200,000,000 (2 billion / 10).

The random number generator will produce integers between (1, 2, ..., 2^31-2) which will be normalized to a real number between 0 and 1. This allows for the generation of Uniform (0,1) random numbers. Most users need to generate random numbers from several different discrete and continuous distributions (e.g., poisson, normal, etc.). It is also necessary that unique instances of random number generators be available in contrast to a global generator which would not provide unique streams for each request of a random number generator.

Statistical gathering

Since most simulations are just computer based statistical sampling experiments, the ability to gather statistics is a necessary requirement of every simulation language. If the results of a stochastic simulation experiment are to have any meaning, appropriate statistical techniques must be used to design and analyze the simulation experiments. There are basically four types of statistics which are minimum requirement for any simulation practitioner; Discrete Time based, Continuous Time Based, Observational based, and Counting. Discrete and Continuous Time based statistics are both statistics that are based on the fraction of time a statistic is at some value. The difference between continuous and discrete time based statistics is that the continuous statistic is continuously changing over time and the discrete statistic is changing in finite discrete jumps over time. Observational based statistics are discrete in that they are generated over some counting value. Count statistics are a discrete statistic generated by counting some occurrence. Taha (1988) provides a good explanation of all but continuous-time based statistics.

Discrete time based statistics --

Discrete-time based statistics are obtained on a particular variable by summing the proportions of time a statistic is at some value. The equations for making this calculation are:

$$mean = \sum_{i=1}^{n} (value_i * fraction\,of\,time_i) \div total\,time$$

$$variance = (\sum_{i=1}^{n} ((value_i - mean\,value)^2 * fraction\,of\,time_i)) \div total\,time$$

It is very important in computer based simulations to minimize the amount of space necessary to store information. One feature that can be utilized to circumvent the need to store all observations in a statistical experiment is: sums of values and sums of squares of values for any statistic. If the only necessary statistical output are means, variances, and confidence intervals, then it is not necessary to save all the statistical observations.

Accumulating sums and sums of square values for each observation will circumvent the need to store all observations when collecting discrete time-based statistics.

$$Sx_i = Sx_{i-1} + X_i * t_i \qquad \text{where } Sx_0 = 0$$
$$SSx_i = SSx_{i-1} + X_i^2 * t_i \qquad \text{where } SSx_0 = 0$$

$$mean = Sx \div total\ time$$
$$variance = (SSx \div total\ time) - (Sx \div total\ time)^2$$

Common simulation statistics that utilize discrete-time based statistics are length of queue and utilization of resources. It should be obvious that these are time based because a queues average length is always defined over some time interval (i.e., the queue has been of size X for n time units) and utilization of resources where utilization is defined as busy time over total time.

Continuous-time based statistics --

Continuous-time based statistics are obtained in a similar manner to discrete-time based statistics except for the way the sums (SXi) of the values and the sums of squares (SSXi) of the values are collected. Since we are dealing with continuously changing variables and not discrete jumps, summing areas is more difficult. Although, there is a simple solution and it utilizes trapezoids to sum areas. Trapezoids are typically used when summing areas under a curve and the equations for generating their statistics are as follows:

$$mean = \sum_{i=1}^{n} (0.5 * (value_i + value_{i-1}) * fraction\ of\ time_i) \div total\ time$$

$$variance = (\sum_{i=1}^{n} ((0.5 * (value_i + value_{i-1}) - mean\ value)^2 * fraction\ of\ time_i)) \div total\ time$$

Since continuous-time based statistics are generated by summing the areas under the curve for that continuous variable, and dividing by the total time, the statistics are available at any time and it is not necessary to keep all of the observations. Accumulating sums and sums of square values for each observation will circumvent the need to store all observations when collecting continuous-time base statistics.

$$Sx_i = Sx_{i-1} + (X_i + X_{i-1}) \div 2 * t_i \qquad \text{where } Sx_0 = 0$$
$$SSx_i = SSx_{i-1} + ((X_i + X_{i-1}) \div 2)^2 * t_i \qquad \text{where } SSx_0 = 0$$

$$mean = Sx \div total\ time$$
$$variance = (SSx \div total\ time) - (Sx \div total\ time)^2$$

Simulations that collect statistics on a continuously changing variable would utilize continuous-time based statistics. They are similar to discrete-time based statistics except the variables are continuously changing over time instead of changing at discrete points in time.

Observational based statistics --

Observational based statistics are obtained on a particular variable by summing the values of each observation and dividing by the number of observations. The equations for making these calculations are:

$$mean = \sum_{i=1}^{n} value_i \div number\ of\ observations$$

$$variance = \sum_{i=1}^{n} (value_i - mean\ value)^2 \div (number\ of\ observations - 1)$$

Since observational based statistics are generated by summing values and dividing by the number of observations, recording an observation is much more simple than time based statistics. Time based statistics require the fraction of time associated with that observation whereas observational based statistics have no time associated with them unless the statistic variable is a measure of time. Accumulating sums and sums of square values for each observation will circumvent the need to store all observations when collecting observational based statistics.

$Sx_i = Sx_{i-1} + X_i$ where $Sx_0 = 0$

$SSx_i = SSx_{i-1} + X_i^2$ where $SSx_0 = 0$

$mean = Sx \div obs$

$variance = (SSx - obs*(Sx \div obs)^2) \div (obs - 1)$

where obs = number of observations

Common simulation statistics that utilize observational based statistics are time in queue and average busy time per server. It should be obvious that these are observational based because the observation is the time that these variables are at these values (i.e., this entity was in the queue for X time units) and busy time of a resources (i.e., this resource was busy for X time units).

Counting statistics --

Counting statistics are referred to as statistics because since most simulations are statistical sampling experiments, counting the occurrences of something is just another statistic collected from the experiment.

$C_i = C_{i-1} + X_i$ where $C_0 = 0$

count = C_i

Common simulation statistics that utilize count statistics are number of customers served and number of times some event occurred.

There are other statistical issues to consider when conducting simulation experiments and they will be briefly mentioned for completeness but their discussion is beyond the scope of this research.

After conducting simulation experiments, the analyst may desire to conduct comparisons of the performance variables to a standard. Paired comparisons and hypothesis are items that will likely occur during the analysis of simulation output. Graphical analysis of data is a necessary requirement of any statistical analysis. Some graphics an analyst may consider are; Box plots, Quantile plots, Histograms, Stem-Leaf plots, X-Y plots, and others. Time series analysis may be required when non-stationary systems are identified. Variance reduction techniques can help in the analysis of simulation output.

Experimental design

Another principal component of a simulation analysis is model experimentation. Experimentation is the process of initializing key parameters in a model and running simulations to make inferences about the behavior of the system being studied. The experimental design, a component of model experimentation, selects a particular approach to gathering the information needed to draw inferences about the system being studied. Since the purpose of most simulation studies is to learn more about the system being studied, carefully designing experiments will return the most information for the least cost.

The first publication on Experimental Design (Fisher, 1935) presented the principal components of a scientific experiment. Since that time many books on experimental design and analysis have appeared (Box and Hunter, 1978), (Petersen, 1985), (Cochran and Cox, 1957), (Heiberger, 1989), (Naylor, 1968), (Diamond, 1989), (Montgomery, 1991).

Experimental design is used in computer simulations to help make decisions, provide estimates of effects, and to determine where desirable responses are obtained. Organizing and analyzing large quantities of data can be difficult. Simulations include parameters (factors), and each value of a parameter is a level. The questions that an engineer or systems analyst may ask are, "what levels and what factors should I evaluate"? "What precision is necessary and how will I present my results in a meaningful way"? It is therefore necessary to be sure of what the objectives are for the simulation analysis before processing by the computer begins. Analyzing statistical output data from a simulation that does not include initial transient (start-up) bias greatly simplifies an analysis.

Full and fractional factorial designs are two traditional methods used to assist in the setup of a multilevel, multifactor experiment. Full factorial designs are more appropriate for a few factors at two levels. Fractional factorial designs would be better for initial simulation screening at the beginning of a simulation analysis which could involve many factors at more than two levels. The Analysis of Variance (ANOVA) and Multivariate Analysis of Variance (MANOVA) are used with full and fractional factorial designs to identify significant factors.

Heuristic techniques are also popular methods of experimental optimization and analysis. Their purpose is to provide some aid or direction in the solution to a problem. There are many heuristics used in computer simulations.

Appendix C: Object oriented concepts

Object oriented analysis and design is more than just a way of programming. It is a way of thinking abstractly about a problem using real world concepts. Traditional programming languages like FORTRAN required the programmer/analyst to think in terms of computer code and not on the abstract application itself. Object oriented modeling and design is very cognitive and thus easily conceptualized. The essence of object oriented analysis reduces to the idea of an object. An object is something that is capable of being seen, touched, or otherwise sensed. Television sets, bicycles, and people are objects that can be seen and touched. Scheduling policies, rules, and criteria are objects that can be sensed. An object has its own identity which makes it unique from every other object. Objects may have much in common with other objects; like people have much in common with other people, and automobiles have much in common with other automobiles, but there is always something that makes each of them different. Object oriented analysis and design stresses conceptualization rather than implementation. Advantages to OOP over traditional procedural programming have been documented in Cox (1986) and Meyer (1987). According to Meyer (1987):

> "...object-oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs."

The OOP paradigm clearly has much to offer to scientific modeling.

There are many implementations of object oriented languages. Fortunately, they have much in common. Object oriented languages provide four key concepts that distinguish them from traditional procedural languages. The key elements of object oriented language are; abstraction, encapsulation, inheritance, and polymorphism. Abstractions are user defined data types (class); inheritance provides hierarchical relationships and code reuse; encapsulation separates the implementation details of an object from its behavior; and polymorphism is the capacity of an object to have several shapes. Many of these ideas originated in the 1960's (Strachey, 1967).

Classes and objects

Before discussing the idea of abstraction and abstract data types (ADT'S), the concept of the class and object must be introduced. An object has a name and the ability to perform operations on itself. The name allows reference to the class definition and an instance of this class creates an object. Objects perform operations on themselves when they are sent messages. These operations are commonly referred to as methods, and these methods answer messages with, in general, another object. The process by which an object responds to a message, whether logic, algorithm, or decision, is unknown to the sender; only a

reply in the form of an object is returned to the sender. The external view of an object with messages and reply's are depicted in figure A 1.

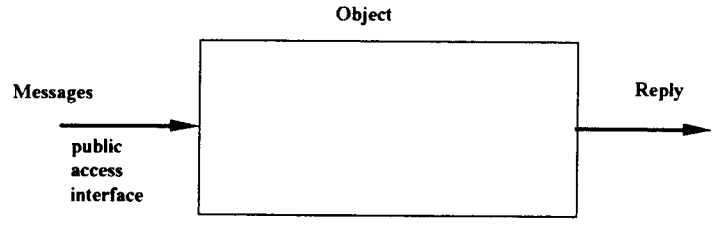


Figure A 1. The external view of an object.

The internal view of an object consists of data and methods, as shown in figure A 2. An object can have data, the values of which are other objects. An objects data is only used by the object itself. The data is not available to other objects unless some public access has been made available. A method is a sequence of instructions that when executed, perform an operation on the object. The instructions for the method typically involve invoking other methods of the object and its data.

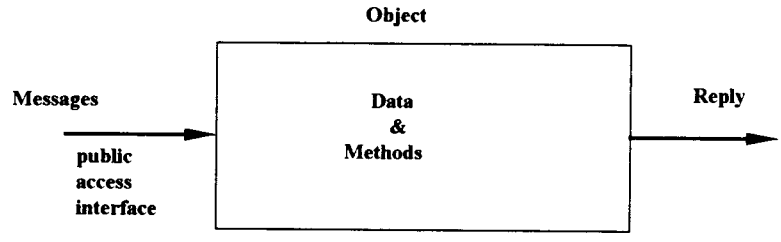


Figure A 2. The internal view of an object.

Several objects may have the same data and methods but each represents a unique instantiation. To define similar but distinct objects, an object is created as an instance of a class, where all instances have the same type of data and methods. Classes are used to create new objects that respond to messages. The relationship between a class and the instances created from it are depicted in figure A 3.

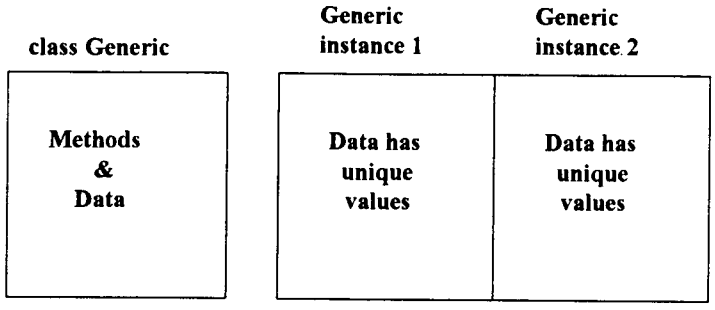


Figure A 3. A class and two instances of the class.

The way in which a class of objects responds to messages is called the behavior of the objects. The specification defines the behavior of the objects by a precise description of the effect of each message. The implementation of a class requires writing the methods needed to provide behavior and listing the data which are other objects which become part of the class. The separation of specification and implementation enhances abstraction because specification defines behavior and implementation defines how the behavior is accomplished. From an external viewpoint, one does not care about the implementation, only the specification.

Abstract classes and data types

An abstract data type (ADT) is defined as an interface to a data abstraction that does not define implementation. Pre-defined data types in the C++ language are called fundamental data types (FDT's). Some FDT's are: void, char, int, float, and double (Borland C++ Programmers Guide, 1992). Objects that are ADT's are instantiated just like FDT's. ADT's are user defined types. These user defined types can consist of ADT's and or FDT's.

```
class Something
{
//-- data --//
float      aTypeOfFloatObject;          // instance of a FDT
Widget     aTypeOfWidgetObject;         // instance of a ADT

//-- methods --//
Something(void);
}
```

Figure A 4. An abstract data type 'Something'.

The two objects in class Something; aTypeOfFloatObject and aTypeOfWidgetObject, are instances of type float and Widget respectively. The interface to these two objects are given as instances, aTypeOfFloatObject and aTypeOfWidgetObject of their respective types. Most programmers are familiar with the interface to FDT's because they are given in compiler references. Thus, we know that we can initialize a 32 bit float to any value between (+or- 3.4 x 10^38). Initialization of an ADT is defined by its constructor. The constructor for an ADT may take zero, one, or more than one argument during the initialization of the object. The Something::Something() constructor takes no arguments but it initializes it's data to the value 0 (Figure A 5).

```
Something::Something(void) :
aTypeOfFloatObject(0.0),
aTypeOfWidgetObject(0.0)
{ }
```

Figure A 5. A class constructor.

Instantiating a class to create an object requires that we run a class constructor. Class constructors can be run statically or dynamically. The static and dynamic creation of 2 FDT's and 2 ADT's are presented in figure A 6.

```
Something somethingInstance;                          // static instance of ADT
Something *somethingInstancePointer = new Something;  // dynamic instance of ADT

float floatInstance(2.0);                             // static instance of FDT
float *floatInstancePointer = new float(4.0);         // dynamic instance of FDT
```

Figure A 6. Static and Dynamic fundamental and abstract data types.

Inheritance and hierarchical relationships

Inheritance implies a hierarchical class relationship which defines a set of classes and their relationship to one another. It is also a reuse mechanism that allows you to create a new class from an existing class and add some added data and behavior. There are two forms of inheritance in C++, public and private.

Public inheritance is used when it is desired to allow the non-private data and methods of the parent or base class to be inherited to the derived or subclass. The derived or subclass will inherit non-private data and methods as if they were its own. Such hierarchies may be many levels deep but are not usually more than three or four in practice.

Inheritance is sometimes implemented as a means to define a new class that is an incremental refinement to its parent class. The parent class does most everything the user wants and the new class slightly modifies the parents implementation. Inheritance can also be used to inherit duel functionality. Multiple inheritance is usually used to implement duel functionality where two different parents provide each of their data and methods to the derived class. A graphical representation of hierarchical relationships are shown in figure A 7.

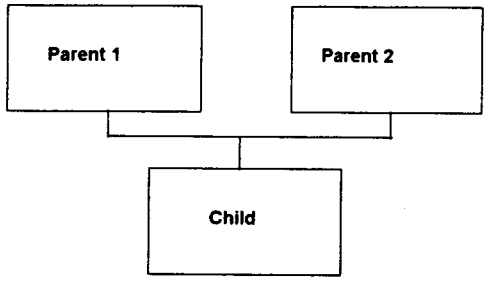


Figure A 7. Hierarchical relationships.

Encapsulation of data and behavior

Encapsulation or information hiding consists of separating the internal implementation details of an object from the accessible external aspects of an object. Encapsulation prevents a program from becoming interdependent on change. The internal implementation of an object can be changed without effecting the application that uses it. Sometimes it is necessary to change an objects internal code for performance improvement or bug fixes (flawed code) but ideally, this should not affect other objects which send messages to this object. Encapsulation of data and behavior into a single entity in OOP languages makes programming cleaner and more powerful than in conventional languages that separate data structure and behavior. Niklaus Wirth wrote a famous equation for a title of a book that summarized the principals of programming (Wirth, 1976):

Algorithms + Data Structures = Programs

In light of that statement and the principals of OOP I would modify Wirth's statement to read:

Algorithms + Data Structures = Objects

Since an object encapsulates data and methods, it must provide a public interface so other objects can send it messages. Thus, the only means of communicating with an object is with the objects public access methods.

Polymorphism and virtual functions

Polymorphism is defined as : having, assuming, or occurring in various forms, characters, or styles (Webster's, 1975). Polymorphism is basically the capacity of an object to have several shapes or several forms. OOP programs utilize polymorphism so that the same operation can take on different forms in different classes. The Update operation for a simulation object would update that objects state at the current time whereas Update for a graphic object could re-draw the object. Dynamic polymorphism is implemented via dynamic binding and inheritance and this technique selects the method to implement at run time. Selecting a method at run time is OOP paradigm that is implemented when a method is declared virtual.

Appendix D: Generic simulation classes

The derived generic simulation classes are simulation objects that can be re-used in many simulation models. The Population class, Queue class, ResourceServer class, Resource class, and Sink class have direct applicability in queuing models but practitioners should not limit their extendibility. Ecosystem simulations that have sources, sinks, producers, and consumers, can also use the generic simulation object templates as a starting point to the development of their simulation objects. The generic simulation objects are shown in figure A 8 at the derived level.

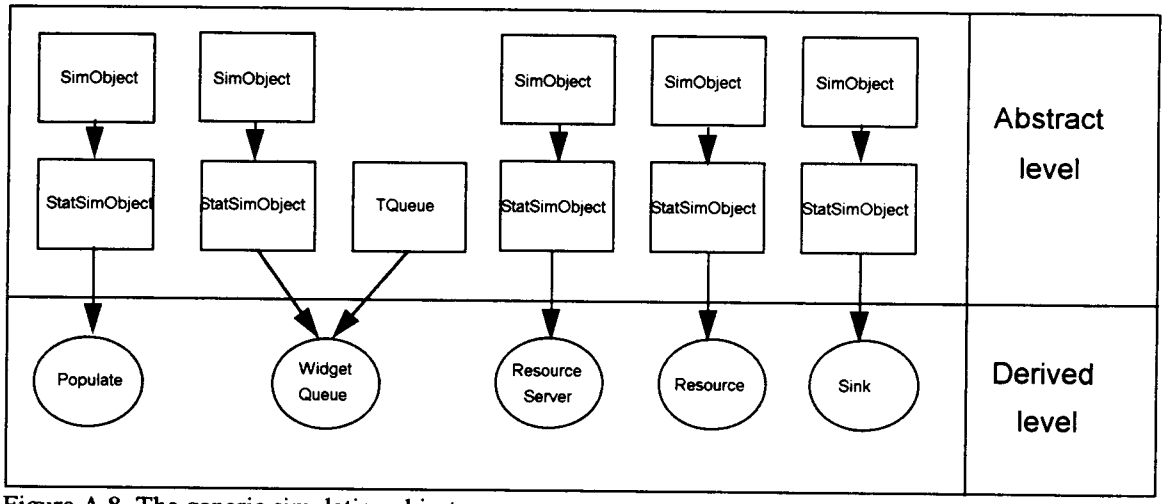


Figure A 8. The generic simulation objects.

The Population object produces Widgets at some inter-arrival rate and injects the Widget objects into the system via the Broadcast-Notify push object transfer methodology. The main intent of the Population object is to produce Widgets and introduce Widgets into the system

The Population object contains only two data members. The pRand data member is a pointer to a random number generator. The interface to this random number generator is kept generic by typing it as its abstract parent class Rand. The data member generateUntilTime establishes the length of time this simulation object will produce Widgets.

The class constructor Population and method Update are the only two methods of the Population object. The constructor of the Population object defines statistics that will be collected during the life of this simulation object. The arrival distribution is also established during the construction of the object. Update contains the logic necessary to create Widget objects and transfer them to the system.

The WidgetQueue class is a container class that queues Widgets on a first in first out (FIFO) basis. Queues are typically located before resources in a flow chart of process flow. Real life examples of queues

are; customers waiting in a bank queue to be serviced by a teller resource, and jobs in a multi-user operating system queuing up for the resource CPU time.

The WidgetQueue object's only data member is the queueCapacity. The queueCapacity is set to limit the amount of Widgets contained within the queue. If the WidgetQueue has reached its holding capacity, it will not accept additional Widget's.

The class constructor WidgetQueue and the Notify method are the only two methods of the WidgetQueue simulation object. WidgetQueues are not simulated so they do not have an Update method. The WidgetQueue accepts and distributes Widget's via the Broadcast-Notify push and Broadcast-Notify-Transfer pull object transfer methodology

The ResourceServer object stores a list of resources and distributes work to idle resources. There is circular communication between the resource server and the resources. Normally, when their are parallel resources, a resource server is used. When resources are serial, or only one, the resource server is not needed. The resource server provides group statistics which would not be available if multiple resources were acting independently.

The ResourceServer class data consists of the number of resourcesScheduled and a ResouceList. The number of resourcesScheduled is defined at construction time for the object. The object will create multiple duplicate resources, as defined by resourcesScheduled data member, and place them in the ResourceList.

The constructor ResourceServer creates the number of resources defined by the resourcesScheduled data member and places these in the ResouceList. Objects are transferred in and out of the ResourceServer object by the push and pull methodologies Broadcast-Notify and Broadcast-Notify-Transfer. When an object enters the ResourceServer, it is routed to a Resource by the StartService method. Objects that have finished service with the Resource objects depart the ResourceServer through the EndService method.

The Resource class provides some service to Widget objects. Resources will let ResourceServers handle notification messages if the Resource object was created by a ResourceServer or they will handle notification messages otherwise.

The Resource class contains three data members; pIsServing, pResourceServer, and pRand. The current object being served is referenced by the pIsServing data member. If a ResourceServer created this Resource, it will be referenced by the pResourceServer data member. Stochastic service times are made available by the pRand data member.

The class constructor Resource initializes the objects data, establishes a statistical collection object, and creates a random number generator object. Update is a required method of this object since it will be simulated through time. Service completion is the event that is registered with the simulation environment when a Widget object begins service. Objects are transferred in and out of the Resource object by the push and pull methodologies Broadcast-Notify and Broadcast-Notify-Transfer.

When a Widget object enters the Resource object, the Resource object will StartService on the Widget object and register its end of service with the simulation environment. EndService marks the departure point for the Widget object, where it is routed to the next accepting destination.

The Sink object is the disposal point for Widget objects. Widget objects are pushed to the Sink object via the push Broadcast-Notify methodology.

The Sink constructor creates a statistical object to collect statistics. The Notify method receives Widgets via the push methodology Broadcast-Notify.

Appendix E: Object oriented simulation examples

The object oriented simulation examples were meant to provide verification of an accurately working simulation engine. The continuous examples utilize the numerical integrator while the stochastic example uses the statistical features of the simulation engine. Two bog ecosystem simulations were conducted to demonstrate the simplicity of modeling a system from two different viewpoints. The first ecosystem was modeled as one unit whereas in the second implementation the system was modeled as interacting units. Both systems produce the same results, with the exception of small insignificant rounding errors. Simulationist will decide which implementation is appropriate for their needs

Continuous deterministic simulation -- Results

```
Results from Pegden used as a standard.

Project: Sample Problem 11.1
Analyst: SM
Replication ended at time: 2.0
```

CONTINUOUS-CHANGE VARIABLES

Indentifier	Average	Variation	Minimum	Maximum	Final Value
Plants	21.798	.30825	.83000	31.884	16.924
Herbivores	.57339	.33659	.30000E-2	.84387	.43784
Carnivores	.53701	.40285	.10000E-3	.79025	.56767
Organic	57.660	.64272	.00000	120.29	120.29
Environment	25.857	.64921	.00000	54.412	54.412
Solar	95.900	.44763	.00000	156.80	95.899

```
# Note that Pegden's minimum for solar is incorrect.


Results from the simulation engine classes.

Simulation: Cedar bog lake ecosystem
Start time: 0.0
End time: 2.0
Statistical reset time: 0.0
```

CONTINUOUS TIME BASED VARIABLES

Object	Statistic	Average	Variation	Min	Max	#Obs	Final Value
Bog	Plants	21.8357	0.3053	0.8300	31.8997	800	16.9244
Bog	Herbivore	0.5744	0.3338	0.0030	0.8444	800	0.4378
Bog	Carnivore	0.5380	0.3995	0.0001	0.7905	800	0.5677
Bog	Organic	57.8061	0.6386	0.0000	120.2911	800	120.2911
Bog	Env	25.9227	0.6451	0.0000	54.4120	800	54.4120
Bog	Solar	95.8988	0.4493	35.0035	156.7965	800	95.9000

Continuous-deterministic -- A one object system

```
Print date: 8/17/1992     Print time:  9:11
//---------------------------------------------------------------------
//  PROGRAM: flags.h
//
//  PURPOSE: Event, statistical ID's, and notifications for simulation
//---------------------------------------------------------------------

#if !defined _FLAGS_H
#define      _FLAGS_H 1


enum STAT_ID_TYPE
   {
   ID_PLANTS,
   ID_HERBIVORES,
   ID_CARNIVORES,
   ID_ORGANIC,
   ID_ENVIRONMENT,
   ID_SOLAR
   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:12
//--------------------------------------------------------------------
//  PROGRAM: simulate.hpp
//
//  PURPOSE: Main program for continuous Bog simulation
//           See Introduction to Simulation Using Siman, pg. 507
//           Sample problem 11.1, Cedar Bog Lake.
//--------------------------------------------------------------------

#if !defined _SIMULATE_HPP
#define      _SIMULATE_HPP 1

// this file is empty at the moment.

#endif
```

```
Print date: 8/17/1992     Print time:  9:12
//---------------------------------------------------------------------
//  PROGRAM: simulate.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _SIMULATE_HPP )
#include "simulate.hpp"
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _BOG_HPP )
#include "bog.hpp"
#endif

#if !defined ( _REPORT_HPP )
#include "report.hpp"
#endif


//----------------------- Global variables ----------------------//

char *gszAppName = "Simulate";
HINSTANCE ghInst = 0;

//------------------------- Constants --------------------------//

const float BOG_TIMESTEP = 0.0025;      // this should be const double
                                        // bug in compiler

int main( int argc, char* argv[ ] )
   {
   SimClock simClock( "Cedar bog lake ecosystem" );

   //---------------- create simulation objects -----------------//

   //-- cedar bog ecosystem --//
   Bog bog( simClock.GetCurrentTime(), BOG_TIMESTEP );
   simClock.RegisterSimObj( &bog );

   //-- debug information --//
   simClock.RegisterEvent( NULL, 0, 0, ET_START_TRACE, &bog );
   simClock.RegisterEvent( NULL, 0.025, 0, ET_STOP_TRACE, &bog );

   simClock.RegisterEvent( NULL, 1.98, 0, ET_START_TRACE, &bog );
   simClock.RegisterEvent( NULL, 1.99, 0, ET_STOP_TRACE, &bog );


   //------------------- set up the simulaton --------------------//

   //-- set a large stop time since the bank defines the end time --//
   simClock.SetStopTime( 2.0 );

   //- run the simulation --//
   simClock.Run();

   //--------------------- generate reports ----------------------//

   MyHeaderOutput( &simClock );

   return 1;
   }
```

```
Print date: 8/17/1992     Print time:  9:12
//-----------------------------------------------------------------
// PROGRAM: bog.hpp
//
// PURPOSE: Cedar bog lake ecosystem class
//
// NOTE: This example was taken from "Introduction to Simulation Using
//       SIMAN by C. Dennis Pegden, Robert E. Shannon, and Randall P.
//       Sadowski, 1990, pgs 507 - 511.
//-----------------------------------------------------------------

#if !defined _BOG_HPP
#define      _BOG_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif


class Bog : public StatSimObj
   {
   protected:
      //-- endogenous state variables --//
      double plants;
      double herbivores;
      double carnivores;
      double organic;
      double environment;

      //-- non-controllable exogenous state variables --//
      double solar;

   public:
      //-- constructor --//
      Bog( double currentTime, double _timeStep );

      //-- property inspector --//
      virtual void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   );

#endif
```

```
Print date: 8/17/1992     Print time:  9:12
//---------------------------------------------------------------------
//  PROGRAM: bog.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//------------------------- Includes --------------------------//

#if !defined ( _BOG_HPP )
#include "bog.hpp"
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif


//-- Bog::Bog() ------------------------------------------------------
//
//-- constructor
//---------------------------------------------------------------------

Bog::Bog( double currentTime, double _timeStep )
   : StatSimObj( "Bog", _timeStep, 0, NULL ),
     plants( 0.83 ),
     herbivores( 0.003 ),
     carnivores( 0.0001 ),
     organic( 0.0 ),
     environment( 0.0 ),
     solar ( 95.9 )

   {
   //-- register statistics --//
   AddStatsContinuousObj( ID_PLANTS, "Plants", currentTime, plants );
   AddStatsContinuousObj( ID_HERBIVORES, "Herbivores", currentTime, herbivores )
   AddStatsContinuousObj( ID_CARNIVORES, "Carnivores", currentTime, carnivores )
   AddStatsContinuousObj( ID_ORGANIC, "Organic", currentTime, organic );
   AddStatsContinuousObj( ID_ENVIRONMENT, "Env", currentTime, environment );
   AddStatsContinuousObj( ID_SOLAR, "Solar", currentTime, solar );
   }


//-- Bog::StartTrace() -----------------------------------------------
//
//-- sets up the trace array for this object
//---------------------------------------------------------------------

void Bog::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is data member of SimObj

   pTraceArray->AddTraceData( "plants",      TYPE_DOUBLE, (void*) &plants );
   pTraceArray->AddTraceData( "herbivores",  TYPE_DOUBLE, (void*) &herbivores
   pTraceArray->AddTraceData( "carnivores",  TYPE_DOUBLE, (void*) &carnivores
   pTraceArray->AddTraceData( "organic",     TYPE_DOUBLE, (void*) &organic );
   pTraceArray->AddTraceData( "environment", TYPE_DOUBLE, (void*) &environmen
   pTraceArray->AddTraceData( "solar",       TYPE_DOUBLE, (void*) &solar );

   return;
   }


//-- Bog::Update() ---------------------------------------------------
//
//-- update this simulation object
//---------------------------------------------------------------------

int Bog::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 5 ];

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = plants;
   stateVar[ 1 ] = herbivores;
   stateVar[ 2 ] = carnivores;
   stateVar[ 3 ] = organic;
   stateVar[ 4 ] = environment;

   //-- integrate --//
   //pSimClock->IntRKS( this, 5, stateVar );
   double time = pSimClock->GetCurrentTime();
   //IntRKS( this, timeStep, time, 5, stateVar );  // virtual function
   IntRKS( this, timeStep, time, 5, stateVar, (STATEPROC) &Bog::State );

   //-- reset after one timeStep --//
   //-- set equivalance --//
   plants      = stateVar[ 0 ];
   herbivores  = stateVar[ 1 ];
   carnivores  = stateVar[ 2 ];
   organic     = stateVar[ 3 ];
   environment = stateVar[ 4 ];

   //-- update statistics --//
   AddContinuousObservation( ID_PLANTS,      plants, time );
   AddContinuousObservation( ID_HERBIVORES,  herbivores, time );
   AddContinuousObservation( ID_CARNIVORES,  carnivores, time );
   AddContinuousObservation( ID_ORGANIC,     organic, time );
   AddContinuousObservation( ID_ENVIRONMENT, environment, time );
   AddContinuousObservation( ID_SOLAR,       solar, time );

   return ES_UPDATE;
   }


//-- Bog::State() ----------------------------------------------------
//
//-- list of differential equations for this object
//---------------------------------------------------------------------

//-- define equivalence --//
```

```
#define plantRate         derive[ 0 ]
#define herbivoreRate     derive[ 1 ]
#define carnivoreRate     derive[ 2 ]
#define organicRate       derive[ 3 ]
#define environmentRate   derive[ 4 ]

void Bog::State( int svCount, double *stateVar, double *derive, double time )
   {
   //-- update exogenous variables --//
   solar = 95.9 * ( 1.0 + 0.635 * sin( 2.0 * M_PI * time ) );

   //-- set equivalence --//
   plants      = stateVar[ 0 ];
   herbivores  = stateVar[ 1 ];
   carnivores  = stateVar[ 2 ];
   organic     = stateVar[ 3 ];
   environment = stateVar[ 4 ];

   //-- update differential equations ( special case of linear system ) --//
   plantRate     = solar - 4.03 * plants;
   herbivoreRate = 0.48 * plants - 17.87 * herbivores;
   carnivoreRate = 4.85 * herbivores - 4.65 * carnivores;
   organicRate   = 2.55 * plants + 6.12 * herbivores + 1.95 * carnivores;
   environmentRate = 1.0 * plants + 6.9 * herbivores + 2.7 * carnivores;

   return;
   }
```

Continuous-deterministic -- A multi-object system

```
Print date: 8/17/1992     Print time:  9:17
//-------------------------------------------------------------------
//  PROGRAM: flags.h
//
//  PURPOSE: Event, statistical ID's, and notifications for simulation
//-------------------------------------------------------------------

#if !defined _FLAGS_H
#define      _FLAGS_H 1


enum STAT_ID_TYPE
   {
   ID_PLANTS,
   ID_HERBIVORES,
   ID_CARNIVORES,
   ID_ORGANIC,
   ID_ENVIRONMENT,
   ID_SOLAR
   };

enum STAT_NT_TYPE
   {
   NT_PLANTS,
   NT_HERBIVORES,
   NT_CARNIVORES,
   NT_ENVIRONMENT,
   NT_ORGANIC
   };



#endif
```

```
Print date: 8/17/1992     Print time:  9:17
//---------------------------------------------------------------
// PROGRAM: date.hpp
//---------------------------------------------------------------

#if !defined _DATE_HPP
#define      _DATE_HPP 1

void PrintTime( void );

#endif
```

```
Print date: 8/17/1992     Print time:  9:19
//---------------------------------------------------------------------
// PROGRAM: date.cpp  (ldate.exe)
//
// PURPOSE: Prints the date and time
//---------------------------------------------------------------------

#include <dos.h>
#include <stdio.h>

#if !defined ( _DATE_HPP )
#include "date.hpp"
#endif


//int main( void );

void PrintTime( void )
   {
//   struct date d;
   struct time t;

//   getdate(&d);
   gettime(&t);

//   printf("Print date: %d", d.da_mon );
//   printf("/%d", d.da_day );
//   printf("/%d", d.da_year );
   printf("     Print time: %2d:%02d:%02d:%02d\n", t.ti_hour, t.ti_min, t.ti_sec

   return;
   }


/*
--------------------------
int main( void )
   {
   PrintDate();

   return 0;
   }
--------------------------
*/
```

```
Print date: 8/17/1992     Print time:  9:19
//--------------------------------------------------------------------
//  PROGRAM: simulate.hpp
//
//  PURPOSE: Main program for continuous Bog simulation
//           See Introduction to Simulation Using Siman, pg. 507
//           Sample problem 11.1, Cedar Bog Lake.
//--------------------------------------------------------------------

#if !defined _SIMULATE_HPP
#define      _SIMULATE_HPP 1

#include <simclock.hpp>
#include <report.hpp>

#include "plants.hpp"
#include "herbivor.hpp"
#include "carnivor.hpp"
#include "organic.hpp"
#include "environ.hpp"

void AddNotificationObjects( SimClock *pSimClock, Plants *pPlants,
      Herbivores *pHerbivores, Carnivores *pCarnivores,
      Organic *pOrganic, Environment *pEnvironment );

#endif
```

```
Print date: 8/17/1992     Print time:  9:18
//-------------------------------------------------------------------
//  PROGRAM: simulate.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _SIMULATE_HPP )
#include "simulate.hpp"
#endif

#if !defined ( _DATE_HPP )
#include "date.hpp"
#endif

//---------------------- Global variables ---------------------//

char *gszAppName = "Simulate";
HINSTANCE ghInst = 0;

//------------------------- Constants -------------------------//

const double TIMESTEP = 0.0025;


//-- AddNotificationObjects() -----------------------------------
//
//-- AddNotifyFilter( pObjectToNotify, notifyType, pBroadcastingObject );
//
//   pObjectToNotify wants to be notified when pBroadcastingObject
//   broadcasts a notifyType message.
//-------------------------------------------------------------------

void AddNotificationObjects( SimClock *pSimClock, Plants *pPlants,
      Herbivores *pHerbivores, Carnivores *pCarnivores,
      Organic *pOrganic, Environment *pEnvironment )
   {

   //-- plants --//
   // Plants want to be notified whenever a NT_PLANTS is sent by,
   // Herbivores, Organic, or Environment.
   SimObjArray simObjFilterPlants;
   simObjFilterPlants.Append( (SimObj*) pHerbivores );
   simObjFilterPlants.Append( (SimObj*) pOrganic );
   simObjFilterPlants.Append( (SimObj*) pEnvironment );
   pSimClock->AddNotifyFilter( (SimObj*) pPlants, NT_PLANTS,
         &simObjFilterPlants );

   //-- herbivores --//
   // Herbivores want to be notified whenever a NT_HERBIVORES is sent by,
   // Carnivores, Organic, or Environment.
   SimObjArray simObjFilterHerbivores;
   simObjFilterHerbivores.Append( (SimObj*) pCarnivores );
   simObjFilterHerbivores.Append( (SimObj*) pOrganic );
   simObjFilterHerbivores.Append( (SimObj*) pEnvironment );
   pSimClock->AddNotifyFilter( (SimObj*) pHerbivores, NT_HERBIVORES,
         &simObjFilterHerbivores );

   //-- carnivores --//
   // Carnivores want to be notified whenever a NT_CARNIVORES is sent by,
   // Organic, or Environment.
   SimObjArray simObjFilterCarnivores;
   simObjFilterCarnivores.Append( (SimObj*) pOrganic );
   simObjFilterCarnivores.Append( (SimObj*) pEnvironment );
   pSimClock->AddNotifyFilter( (SimObj*) pCarnivores, NT_CARNIVORES,
         &simObjFilterCarnivores );

   return;
   }


int main( int argc, char* argv[ ] )
   {
   SimClock simClock( "Cedar bog lake ecosystem" );

   //-- create Plants object --//
   Plants plants( simClock.GetCurrentTime(), TIMESTEP );
   simClock.RegisterSimObj( &plants );

   //-- create Herbivores object --//
   Herbivores herbivores( simClock.GetCurrentTime(), TIMESTEP );
   simClock.RegisterSimObj( &herbivores );

   //-- create Carnivores object --//
   Carnivores carnivores( simClock.GetCurrentTime(), TIMESTEP );
   simClock.RegisterSimObj( &carnivores );

   //-- create Organic object --//
   Organic organic( simClock.GetCurrentTime(), TIMESTEP );
   simClock.RegisterSimObj( &organic );

   //-- create Environment object --//
   Environment environment( simClock.GetCurrentTime(), TIMESTEP );
   simClock.RegisterSimObj( &environment );

   //-- debug information --//
//   simClock.RegisterEvent( NULL, 0, 0, ET_START_TRACE, NULL );
//   simClock.RegisterEvent( NULL, 0.025, 0, ET_STOP_TRACE, NULL );

//   simClock.RegisterEvent( NULL, 1.98, 0, ET_START_TRACE, NULL );
//   simClock.RegisterEvent( NULL, 1.99, 0, ET_STOP_TRACE, NULL );


   //------------------- set up the simulaton --------------------//

   //-- 2 year stop time --//
   simClock.SetStopTime( 100.0 );

   //-- set up object communication --//
   AddNotificationObjects( &simClock, &plants, &herbivores, &carnivores,
         &organic, &environment );

   //-- give the trace output function --//
//   simClock.SetTraceProc( (TRACEPROC) &TraceOutput );

   //- run the simulation --//
   PrintTime();
```

```
simClock.Run();
PrintTime();
//-------------------- generate reports ----------------------//

MyHeaderOutput( &simClock );

return 1;
}
```

```
Print date: 8/17/1992     Print time:  9:18
//-----------------------------------------------------------------
// PROGRAM: plants.hpp
//-----------------------------------------------------------------

#if !defined _PLANTS_HPP
#define      _PLANTS_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif


class Plants : public StatSimObj
   {
   protected:
      double plants;

      //-- endogenous state variables --//
      double solar;

   public:
      //-- constructor --//
      Plants( double currentTime, double _timeStep );

      void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      virtual UINT Notify( SimClock *pSimClock, NOTIFY *pNotify );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  9:17
//-------------------------------------------------------------------
//  PROGRAM: plants.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _PLANTS_HPP )
#include "plants.hpp"
#endif

//--------------------------- Defines ---------------------------//

#define plantRate derive[ 0 ]


//-- constructor --//
Plants::Plants( double currentTime, double _timeStep )
   : StatSimObj( "Plants", _timeStep, 0, NULL ),
     plants( 0.83 ),
     solar( 95.9 )
   {
   AddStatsContinuousObj( ID_PLANTS, "Plants", currentTime, plants );
   AddStatsContinuousObj( ID_SOLAR, "Solar", currentTime, solar );
   }

void Plants::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is data
                                  // member of SimObj

   pTraceArray->AddTraceData( "plants", TYPE_DOUBLE,
         (void*) &plants );
   pTraceArray->AddTraceData( "solar",  TYPE_DOUBLE,
         (void*) &solar );

   return;
   }

//-- simulation update --//
int Plants::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 1 ];

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = plants;

   IntRKS( this, timeStep, pSimClock->GetCurrentTime(), 1, stateVar );

   plants = *stateVar;

   AddContinuousObservation( ID_PLANTS, plants,
         pSimClock->GetCurrentTime() );
   AddContinuousObservation( ID_SOLAR, solar,
         pSimClock->GetCurrentTime() );

   return ES_UPDATE;
   }


UINT Plants::Notify( SimClock *pSimClock, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_PLANTS:
         {
         *pNotify->pData = (void*)&plants;
         return TRUE;
         }

      default:
         {
         cout << "Error in Plants::Notify\n";
         return FALSE;
         }
      }  // end of SWITCH
   }


//-- differential equations --//
void Plants::State( int svCount, double *stateVar, double *derive, double tim
   {
   //-- update exogenous variables --//
   solar = 95.9 * ( 1.0 + 0.635 * sin( 2.0 * M_PI * time ) );

   plantRate = solar - 4.03 * stateVar[ 0 ];

   return;
   }
```

```
Print date: 8/17/1992     Print time:  9:20
//-----------------------------------------------------------------
// PROGRAM: herbivor.hpp
//-----------------------------------------------------------------

#if !defined _HERBIVOR_HPP
#define      _HERBIVOR_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif


class Herbivores : public StatSimObj
   {
   protected:
      double herbivores;

   public:
      //-- constructor --//
      Herbivores( double currentTime, double _timeStep );

      void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      virtual UINT Notify( SimClock *pSimClock, NOTIFY *pNotify );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:18
//---------------------------------------------------------------------
//  PROGRAM: herbivor.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//-------------------------- Includes -------------------------//

#if !defined ( _HERBIVOR_HPP )
#include "herbivor.hpp"
#endif

//-------------------------- Defines --------------------------//

#define herbivoreRate derive[ 0 ]


//-- constructor --//
Herbivores::Herbivores( double currentTime, double _timeStep )
   : StatSimObj( "Herbivores", _timeStep, 0, NULL ),
     herbivores( 0.003 )
   {
   AddStatsContinuousObj( ID_HERBIVORES, "Herbivores",
         currentTime, herbivores );
   }


void Herbivores::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is data
                                  // member of SimObj

   pTraceArray->AddTraceData( "herbivores",  TYPE_DOUBLE,
         (void*) &herbivores );

   return;
   }

//-- simulation update --//
int Herbivores::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 2 ];
   double *pStateVar1 = NULL;
   void **ptpStateVar1 = &(void*)pStateVar1;

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = herbivores;
   pSimClock->Broadcast( (SimObj*) this, NT_PLANTS, ptpStateVar1 );

   stateVar[ 1 ] = *((double*) (*ptpStateVar1));
   IntRKS( this, timeStep, pSimClock->GetCurrentTime(), 1, stateVar );

   herbivores = *stateVar;

   AddContinuousObservation( ID_HERBIVORES, herbivores,
         pSimClock->GetCurrentTime() );

   return ES_UPDATE;
   }


UINT Herbivores::Notify( SimClock *pSimClock, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_HERBIVORES:
         {
         *pNotify->pData = (void*)&herbivores;
         return TRUE;
         }

      default:
         {
         cout << "Error in Herbivores::Notify\n";
         return FALSE;
         }
      }  // end of SWITCH
   }

//-- differential equations --//
void Herbivores::State( int svCount, double *stateVar, double *derive, double
   {
   herbivoreRate = 0.48 * stateVar[ 1 ] - 17.87 * stateVar[ 0 ];

   return;
   }
```

```
Print date: 8/17/1992     Print time:  9:17
//-----------------------------------------------------------------
// PROGRAM: carnivor.hpp
//-----------------------------------------------------------------

#if !defined _CARNIVOR_HPP
#define      _CARNIVOR_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif


class Carnivores : public StatSimObj
   {
   protected:
      double carnivores;

   public:
      //-- constructor --//
      Carnivores( double currentTime, double _timeStep );

      void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      virtual UINT Notify( SimClock *pSimClock, NOTIFY *pNotify );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:20
//-----------------------------------------------------------------
//  PROGRAM: carnivor.cpp
//-----------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _CARNIVOR_HPP )
#include "carnivor.hpp"
#endif

//--------------------------- Defines --------------------------//

#define carnivoreRate derive[ 0 ]


//-- constructor --//
Carnivores::Carnivores( double currentTime, double _timeStep )
   : StatSimObj( "Carnivore", _timeStep, 0, NULL ),
     carnivores( 0.0001 )
   {
   AddStatsContinuousObj( ID_CARNIVORES, "Carnivores",
         currentTime, carnivores );
   }


void Carnivores::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is
                                  // data member of SimObj

   pTraceArray->AddTraceData( "carnivores",  TYPE_DOUBLE,
         (void*) &carnivores );

   return;
   }


//-- simulation update --//
int Carnivores::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 2 ];
   double *pStateVar1 = NULL;
   void **ptpStateVar1 = &(void*)pStateVar1;

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = carnivores;
   pSimClock->Broadcast( (SimObj*) this, NT_HERBIVORES, ptpStateVar1 );

   stateVar[ 1 ] = *((double*) (*ptpStateVar1));
   IntRKS( this, timeStep, pSimClock->GetCurrentTime(), 1, stateVar );

   carnivores = *stateVar;

   AddContinuousObservation( ID_CARNIVORES, carnivores,
         pSimClock->GetCurrentTime() );

   return ES_UPDATE;
   }


UINT Carnivores::Notify( SimClock *pSimClock, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_CARNIVORES:
         {
         *pNotify->pData = (void*)&carnivores;
         return TRUE;
         }

      default:
         {
         cout << "Error in Carnivores::Notify\n";
         return FALSE;
         }
      }  // end of SWITCH
   }


//-- differential equations --//
void Carnivores::State( int svCount, double *stateVar, double *derive, double
   {
   carnivoreRate = 4.85 * stateVar[ 1 ] - 4.65 * stateVar[ 0 ];

   return;
   }
```

```
Print date: 8/17/1992     Print time:  9:20
//-----------------------------------------------------------------
// PROGRAM: organic.hpp
//-----------------------------------------------------------------

#if !defined _ORGANIC_HPP
#define      _ORGANIC_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif


class Organic : public StatSimObj
   {
   protected:
      double organic;

   public:
      //-- constructor --//
      Organic( double currentTime, double _timeStep );

      void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      virtual UINT Notify( SimClock *pSimClock, NOTIFY *pNotify );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   );

#endif
```

```
Print date: 8/17/1992     Print time:  9:17
//---------------------------------------------------------------------
//  PROGRAM: organic.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _ORGANIC_HPP )
#include "organic.hpp"
#endif

//-------------------------- Defines ----------------------------//

#define organicRate derive[ 0 ]


//-- constructor --//
Organic::Organic( double currentTime, double _timeStep )
   : StatSimObj( "Organic", _timeStep, 0, NULL ),
     organic( 0.0 )
   {
   AddStatsContinuousObj( ID_ORGANIC, "Organic",
         currentTime, organic );
   }

void Organic::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is data
                                  // member of SimObj

   pTraceArray->AddTraceData( "organic", TYPE_DOUBLE,
         (void*) &organic );

   return;
   }


//-- simulation update --//
int Organic::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 4 ];
   double *pStateVar1 = NULL;
   void **ptpStateVar1 = &(void*)pStateVar1;
   double *pStateVar2 = NULL;
   void **ptpStateVar2 = &(void*)pStateVar2;
   double *pStateVar3 = NULL;
   void **ptpStateVar3 = &(void*)pStateVar3;

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = organic;
   BOOL retVal;
   retVal = pSimClock->Broadcast( (SimObj*) this, NT_PLANTS,     ptpStateVar1 );
   retVal = pSimClock->Broadcast( (SimObj*) this, NT_HERBIVORES, ptpStateVar2 );
   retVal = pSimClock->Broadcast( (SimObj*) this, NT_CARNIVORES, ptpStateVar3

   stateVar[ 1 ] = *((double*) (*ptpStateVar1));
   stateVar[ 2 ] = *((double*) (*ptpStateVar2));
   stateVar[ 3 ] = *((double*) (*ptpStateVar3));
   IntRKS( this, timeStep, pSimClock->GetCurrentTime(), 1, stateVar );

   organic = *stateVar;

   AddContinuousObservation( ID_ORGANIC, organic,
         pSimClock->GetCurrentTime() );

   return ES_UPDATE;
   }


UINT Organic::Notify( SimClock *pSimClock, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_ORGANIC:
         {
         *pNotify->pData = (void*)&organic;
         return TRUE;
         }

      default:
         {
         cout << "Error in Organic::Notify\n";
         return FALSE;
         }
      }  // end of SWITCH
   }


//-- differential equations --//
void Organic::State( int svCount, double *stateVar, double *derive, double tin
   {
   organicRate = 2.55 * stateVar[ 1 ] + 6.12 * stateVar[ 2 ] +
                 1.95 * stateVar[ 3 ];

   return;
   }
```

```
Print date: 8/17/1992     Print time:  9:17
//-----------------------------------------------------------------
// PROGRAM: environ.hpp
//-----------------------------------------------------------------

#if !defined _ENVIRON_HPP
#define      _ENVIRON_HPP 1

//-------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _SIMCLOCK_HPP )
#include <simclock.hpp>
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif


class Environment : public StatSimObj
   {
   protected:
      double environment;

   public:
      //-- constructor --//
      Environment( double currentTime, double _timeStep );

      void StartTrace( void );

      //-- simulation update --//
      virtual int Update( SimClock *pSimClock, UINT eventType, void *pData );

      virtual UINT Notify( SimClock *pSimClock, NOTIFY *pNotify );

      //-- differential equations --//
      void State( int svCount, double *stateVar, double *derive, double time );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:21
//-------------------------------------------------------------------
//  PROGRAM: environ.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _ENVIRON_HPP )
#include "environ.hpp"
#endif

//--------------------------- Defines ---------------------------//

#define environmentRate derive[ 0 ]

//-- constructor --//
Environment::Environment( double currentTime, double _timeStep )
   : StatSimObj( "Env", _timeStep, 0, NULL ),
     environment( 0.0 )
   {
   AddStatsContinuousObj( ID_ENVIRONMENT, "Env",
        currentTime, environment );
   }

void Environment::StartTrace( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   pTraceArray = new TraceArray;  // pTraceArray is data
                                  // member of SimObj

   pTraceArray->AddTraceData( "environment", TYPE_DOUBLE,
        (void*) &environment );

   return;
   }


//-- simulation update --//
int Environment::Update( SimClock *pSimClock, UINT eventType, void *pData )
   {
   double stateVar[ 4 ];
   double *pStateVar1 = NULL;
   void **ptpStateVar1 = &(void*)pStateVar1;
   double *pStateVar2 = NULL;
   void **ptpStateVar2 = &(void*)pStateVar2;
   double *pStateVar3 = NULL;
   void **ptpStateVar3 = &(void*)pStateVar3;

   //-- stateVar array is the standard interface to the Integrator --//
   //-- set equivalance --//
   stateVar[ 0 ] = environment;
   pSimClock->Broadcast( (SimObj*) this, NT_PLANTS,     ptpStateVar1 );
   pSimClock->Broadcast( (SimObj*) this, NT_HERBIVORES, ptpStateVar2 );
   pSimClock->Broadcast( (SimObj*) this, NT_CARNIVORES, ptpStateVar3 );

   stateVar[ 1 ] = *((double*) (*ptpStateVar1));
   stateVar[ 2 ] = *((double*) (*ptpStateVar2));
   stateVar[ 3 ] = *((double*) (*ptpStateVar3));
   IntRKS( this, timeStep, pSimClock->GetCurrentTime(), 1, stateVar );

   environment = *stateVar;

   AddContinuousObservation( ID_ENVIRONMENT, environment,
        pSimClock->GetCurrentTime() );

   return ES_UPDATE;
   }

UINT Environment::Notify( SimClock *pSimClock, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_ENVIRONMENT:
         {
         *pNotify->pData = (void*)&environment;
         return TRUE;
         }

      default:
         {
         cout << "Error in Environment::Notify\n";
         return FALSE;
         }
      } // end of SWITCH
   }


//-- differential equations --//
void Environment::State( int svCount, double *stateVar, double *derive, double
   {
   environmentRate = 1.0 * stateVar[ 1 ] + 6.9 * stateVar[ 2 ] +
                     2.7 * stateVar[ 3 ];

   return;
   }
```

Discrete-stochastic simulation -- A multi-server queuing system

Multiple Run Global Statistics
Replication Method

Number of replications (obs): 100

Simulation: Queueing simulation: Exp #1
Start time: 0
End time: 484.3
Statistical reset time: 0

(PMMCG)
Xi = a*(Xi-1) mod m

DISTRIBUTIONS

Object	Statistic	Dist	A	B	C	Seed	Ave #Obs	Multiplier
Pop	ArrivRate	Expon	1	-	-	1000000	481	950706376
Res 0	ServeRate	Expon	4	-	-	2000000	108	950706376
Res 1	ServeRate	Expon	4	-	-	3000000	102	950706376
Res 2	ServeRate	Expon	4	-	-	4000000	98	950706376
Res 3	ServeRate	Expon	4	-	-	5000000	89	950706376
Res 4	ServeRate	Expon	4	-	-	6000000	82	950706376

OBSERVATIONAL VARIABLES

Object	Statistic	Average	Standard deviation	Min	Max	Ave #Obs	95% Con-Limit Lower	Upper
WidQueue	Time all	2.3408	1.4499	0.0000	32.5512	480	2.0530	2.6286
WidQueue	Time wait	4.0043	1.7838	0.0001	32.5512	268	3.6502	4.3583
ResServ	Busy	3.9289	0.1615	0.0000	5.0000	960	3.8969	3.9610
ResServ	Idle	1.0711	0.1615	0.0000	5.0000	960	1.0390	1.1031
Sink	Time sys	6.3794	1.5544	0.0001	56.6799	480	6.0709	6.6880

DISCRETE TIME BASED VARIABLES

Object	Statistic	Average	Standard deviation	Min	Max	Ave #Obs	95% Con-Limit Lower	Upper
WidQueue	Len all	2.3668	1.5635	0.0000	37.0000	480	2.0564	2.6771
WidQueue	Len wait	2.9347	1.4768	1.0000	37.0000	268	2.6416	3.2278
Res 0	Util	0.8925	0.0303	0.0000	1.0000	216	0.8865	0.8986
Res 1	Util	0.8562	0.0396	0.0000	1.0000	204	0.8484	0.8641
Res 2	Util	0.8141	0.0499	0.0000	1.0000	197	0.8042	0.8240
Res 3	Util	0.7589	0.0650	0.0000	1.0000	179	0.7460	0.7718
Res 4	Util	0.6952	0.0868	0.0000	1.0000	164	0.6780	0.7124
ResServ	Util	0.8307	0.0451	0.2000	1.0000	480	0.8218	0.8397

COUNT VARIABLES

Object	Statistic	Count	Ave #Obs
Pop	Widg balk	0	0
Pop	Widg crea	480	480

```
Print date: 8/17/1992     Print time:  9:05
//-------------------------------------------------------------------
//  PROGRAM: flags.h
//
//  PURPOSE: Event, statistical ID's, and notifications for simulation
//-------------------------------------------------------------------

#if !defined _FLAGS_H
#define      _FLAGS_H 1


//-- event types --//
enum EVENT_TYPE
   {
   ET_CREATE_WIDGET,
   ET_DEPART
   };

enum STAT_ID_TYPE
   {
   ID_BALK_COUNT,
   ID_TIME_IN_SYSTEM,
   ID_TIME_IN_QUEUE_ALL,
   ID_TIME_IN_QUEUE_WAIT,
   ID_LENGTH_OF_QUEUE_ALL,
   ID_LENGTH_OF_QUEUE_WAIT,
   ID_CUMM_RESOURCE_UTIL,
   ID_RESOURCE_UTIL,
   ID_BUSY_NUMBER,
   ID_IDLE_NUMBER,
   ID_WIDGET_COUNT
   };

enum NOTIFY_TYPE
   {
   NT_QUEUE,
   NT_TRANSFER,
   NT_INIT_TRANSFER,
   NT_IDLE,
   NT_DEPART
   };


#endif
```

```
Print date: 8/17/1992     Print time:  9:05
//-----------------------------------------------------------------
//  PROGRAM: simulate.hpp
//
//  PURPOSE: Main program for discrete event simulation
//           See Simulation Modeling & Analysis by Averill Law and
//           David Kelton, 1991, pg. 524
//-----------------------------------------------------------------

#if !defined _SIMULATE_HPP
#define      _SIMULATE_HPP 1

//----------------------- Forward references --------------------//

class SimEnv;
class Population;
class WidgetQueue;
class ResourceServer;
class Sink;


void AddNotificationObjects( SimEnv *pSimEnv, Population *pPopulation,
      WidgetQueue *pWidgetQueue, ResourceServer *pResourceServer,
      Sink pSink );

void ExperimentOne( SimEnv *pSimEnv );
void ExperimentTwo( SimEnv *pSimEnv );


#endif
```

```
Print date: 8/17/1992     Print time:  9:07
//---------------------------------------------------------------------
//  PROGRAM: simulate.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes -------------------------//

#if !defined ( _SIMULATE_HPP )
#include "simulate.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _POPULATE_HPP )
#include "populate.hpp"
#endif

#if !defined ( _WQUEUE_HPP )
#include "wqueue.hpp"
#endif

#if !defined ( _RSERVER_HPP )
#include "rserver.hpp"
#endif

#if !defined ( _SINK_HPP )
#include "sink.hpp"
#endif

#if !defined ( _REPORT_HPP )
#include <report.hpp>
#endif

#if !defined ( _LOGWND_HPP )
#include <logwnd.hpp>
#endif

#if !defined ( __LIMITS_H )
extern "C"
{
#include <limits.h>
}
#endif


#include <stdlib.h>
#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif

//--------------------------- Constants ------------------------//
```

```
const int RESOURCE_COUNT = 5;          // 5 parallel servers
const float SIMULATION_LENGTH = 480;   // minutes
const float ARRIVAL_RATE = 1.0;        // population arrival rate
const float SERVICE_RATE = 4.0;        // server service rate

//----------------------- Global variables ----------------------//

char *gszAppName = "Simulate";
HINSTANCE ghInst = 0;

LogWindow       *gpLogWnd   = NULL;
LogWindow       *gpTraceWnd = NULL;

//-- AddNotificationObjects() ---------------------------------------
//
//-- AddNotifyFilter( pObjectToNotify, notifyType, pBroadcastingObject );
//
//   pObjectToNotify wants to be notified when pBroadcastingObject
//   broadcasts a notifyType message.
//---------------------------------------------------------------------

void AddNotificationObjects( SimEnv *pSimEnv, Population *pPopulation,
      WidgetQueue *pWidgetQueue, ResourceServer *pResourceServer,
      Sink *pSink )
   {

   //-- The WidgetQueue wants to be notified when the Population
   //   broadcasts a NT_TRANSFER
   pSimEnv->AddNotifyFilter( (SimObj*) pWidgetQueue,
         NT_TRANSFER, (SimObj*) pPopulation );

   //-- The WidgetQueue wants to be notified when the ResourceServer
   //   broadcasts a NT_INIT_TRANSFER
   pSimEnv->AddNotifyFilter( (SimObj*) pWidgetQueue,
         NT_INIT_TRANSFER, pResourceServer );

   //-- The ResourceServer wants to be notified when the WidgetQueue
   //   broadcasts a NT_TRANSFER
   pSimEnv->AddNotifyFilter( (SimObj*) pResourceServer,
         NT_TRANSFER, pWidgetQueue );

   //-- The Sink wants to be notified when the ResourceServer
   //   broadcasts a NT_TRANSFER
   pSimEnv->AddNotifyFilter( (SimObj*) pSink,
         NT_TRANSFER, pResourceServer );

   return;
   }


int main( int argc, char* argv[ ] )
   {
#if defined _Windows && defined EASYWIN
   gpLogWnd = new LogWindow( 0 );
   gpTraceWnd = new LogWindow( 0 );
#endif

   int reps = 10;
```

```
    if ( argc > 1 )
       reps = atoi( argv[ 1 ] );

    cout << "Reps: " << reps;

    //-- Experiment one --//
    StatSimEnv SimEnv1( "Queueing simulation: Exp #1" );
    SimEnv1.SetStopTime( 1000 );

    //-- prints out statistics --//
//    SimEnv1.SetReportProc( MyHeaderOutput );
    SimEnv1.SetGlobalReportProc( MyGlobalHeaderOutput );

    SimEnv1.Replicate( reps, ExperimentOne );


/*
------------
    //-- Experiment two --//
    SimEnv SimEnv2( "Queueing simulation: Exp #2" );
    SimEnv2.SetStopTime( 1000 );
    SimEnv2.SetReportProc( MyHeaderOutput );
    SimEnv2.Replicate( 2, ExperimentTwo );

    //-- prints out global statistics --//
//    MyHeaderOutput( &SimEnv2 );
------------
*/

    if ( gpLogWnd )   delete gpLogWnd;

    if ( gpTraceWnd ) delete gpTraceWnd;

    return 1;
    }


void ExperimentOne( SimEnv *pSimEnv )
    {
    //---------------- create Population object -----------------//

    //-- generate widgets for SIMULATION_LENGTH minutes --//
    Population *pPopulation = new Population( ARRIVAL_RATE, SIMULATION_LENGTH );

    //-- register the population object with this SimEnv --//
    pSimEnv->RegisterSimObj( (SimObj*) pPopulation, RE_NOREGISTER );

    //-- start generating widgets at time 0.0 --//
    pSimEnv->RegisterEvent( ( SimObj* ) pPopulation, 0.0, 0,
          ET_CREATE_WIDGET, NULL );

    //---------------- create WidgetQueue object ----------------//

    //-- create the WidgetQueue SimObj --//
    WidgetQueue *pWidgetQueue = new WidgetQueue( pSimEnv->GetCurrentTime(),
          LONG_MAX );

    //-- register the WidgetQueue object with this SimEnv --//
    pSimEnv->RegisterSimObj( (SimObj*) pWidgetQueue, RE_NOREGISTER );

    //---------------- create ResourceServer object -------------//

    //-- create the ResourceServer SimObj --//
    ResourceServer *pResourceServer = new ResourceServer( pSimEnv,
          (UINT) RESOURCE_COUNT, (double) SERVICE_RATE );

    //-- register the ResourceServer object with this SimEnv --//
    pSimEnv->RegisterSimObj( (SimObj*) pResourceServer, RE_NOREGISTER );

    //------------------- create Sink object -------------------//

    //-- create the sink --//
    Sink *pSink = new Sink;

    //-- register the Sink object with this SimEnv --//
    pSimEnv->RegisterSimObj( (SimObj*) pSink, RE_NOREGISTER );


    //------------------ set up the simulaton -------------------//

    //-- set up object communication --//
    AddNotificationObjects( pSimEnv, pPopulation, pWidgetQueue,
          pResourceServer, pSink );

    //-- register a system event --//
//    pSimEnv->RegisterEvent( NULL, (SIMULATION_LENGTH / 2), 0,
//          ET_ENUM_EVENTLIST, ViewEventList );

    //-- clear bias statistics --//
//    pSimEnv->RegisterEvent( NULL, 60, 0, ET_CLEARSTATS, NULL );

    //- run the simulation --//
    pSimEnv->Run();


    //-- clean up objects --//
    delete pWidgetQueue;
    delete pPopulation;
    delete pResourceServer;
    delete pSink;

    return;

    }


void ExperimentTwo( StatSimEnv *pSimEnv )
    {
    //---------------- create Population object -----------------//

    //-- generate widgets for SIMULATION_LENGTH minutes --//
    Population *pPopulation = new Population( ARRIVAL_RATE, SIMULATION_LENGTH )

    //-- register the population object with this SimEnv --//
    pSimEnv->RegisterSimObj( (SimObj*) pPopulation, RE_NOREGISTER );

    //-- start generating widgets at time 0.0 --//
    pSimEnv->RegisterEvent( ( SimObj* ) pPopulation, 0.0, 0,
          ET_CREATE_WIDGET, NULL );
```

```
//----------------- create WidgetQueue object -----------------//

//-- create the WidgetQueue SimObj --//
WidgetQueue *pWidgetQueue = new WidgetQueue( pSimEnv->GetCurrentTime(),
      LONG_MAX );

//-- register the WidgetQueue object with this SimEnv --//
pSimEnv->RegisterSimObj( (SimObj*) pWidgetQueue, RE_NOREGISTER );

//----------------- create ResourceServer object -------------//

//-- create the ResourceServer SimObj --//
ResourceServer *pResourceServer = new ResourceServer( pSimEnv,
      (UINT) RESOURCE_COUNT, (double) SERVICE_RATE );

//-- register the ResourceServer object with this SimEnv --//
pSimEnv->RegisterSimObj( (SimObj*) pResourceServer, RE_NOREGISTER );

//--------------------- create Sink object --------------------//

//-- create the sink --//
Sink *pSink = new Sink;

//-- register the Sink object with this SimEnv --//
pSimEnv->RegisterSimObj( (SimObj*) pSink, RE_NOREGISTER );


//------------------- set up the simulaton --------------------//

//-- set up object communication --//
AddNotificationObjects( pSimEnv, pPopulation, pWidgetQueue,
      pResourceServer, pSink );

//-- register a system event --//
pSimEnv->RegisterEvent( NULL, (SIMULATION_LENGTH / 2), 0,
      ET_ENUMEVENTLIST, ViewEventList );

//- run the simulation --//
pSimEnv->Run();

//-- clean up objects --//
delete pWidgetQueue;
delete pPopulation;
delete pResourceServer;
delete pSink;

return;

}
```

```
Print date: 8/17/1992     Print time:  9:05
//-------------------------------------------------------------------
//  PROGRAM: widget.hpp
//
//  PURPOSE: An un-named article considered for purposes of
//           hypothetical example.
//-------------------------------------------------------------------

#if !defined _WIDGET_HPP
#define      _WIDGET_HPP 1

class Widget
   {
   private:
      double arrivalTime;

   public:
      //-- constructor --//
      Widget( double _arrivalTime )
         : arrivalTime( _arrivalTime )
         { }

      double GetArrivalTime( void ) { return arrivalTime; }

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:06
//-------------------------------------------------------------------
//  PROGRAM: populate.hpp
//
//  PURPOSE: Population object that creates widgets
//-------------------------------------------------------------------

#if !defined _POPULATE_HPP
#define      _POPULATE_HPP 1

//--------------------------- Includes -------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif


class Population : public StatSimObj
   {
   protected:
      //-- arrival rate random variable --//
      Rand *pRand;

      //-- generate population until --//
      double generateUntilTime;

   public:
      //-- constructor --//
      Population( double arrivalRate, double _generateUntilTime );

      //-- simulation update --//
      virtual int Update( SimEnv *pSimEnv, UINT eventType, void *pData );

   );

#endif
```

```
Print date: 8/17/1992     Print time:  9:07
//-------------------------------------------------------------------
//  PROGRAM: populate.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _POPULATE_HPP )
#include "populate.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif

#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif


//-- Population::Population() ---------------------------------------
//
//-- constructor
//-------------------------------------------------------------------

Population::Population( double arrivalRate, double _generateUntilTime )
   : StatSimObj( "Pop", 1, 0, NULL ),
     pRand( NULL ),
     generateUntilTime( _generateUntilTime )
   {
   //-- statistics --//
   AddStatsCountObj( ID_BALK_COUNT, "Widg balk" );
   AddStatsCountObj( ID_WIDGET_COUNT, "Widg creat" );

   //-- random number generator --//
   pRand = MakeRandExponential( arrivalRate, 0, "ArrivRate" );
   pRand->SetSeed( 200000001 );
   }


//-- Population::Update() -------------------------------------------
//
//-- update this simulation object
//-------------------------------------------------------------------

int Population::Update( SimEnv *pSimClock, UINT eventType, void *pData )
{
timeStep = pRand->RandValue(); // arrival rate

//-- check population generation stop time --//
if( ( timeStep + pSimClock->GetCurrentTime() ) > generateUntilTime )
   return( ES_NOUPDATE ); // stop generating customers

switch( eventType )
   {
   case ET_CREATE_WIDGET:
      {
      if( notify )          // someone will receive this customer
         {
         Widget *pWidget = new Widget( pSimClock->GetCurrentTime() );
         AddCountObservation( ID_WIDGET_COUNT, 1 );

         BOOL received = pSimClock->Broadcast( (SimObj*) this,\
            NT_TRANSFER, &(void*)pWidget );

         if( ! received )
            {
            AddCountObservation( ID_BALK_COUNT, 1 );
            delete pWidget;
            }
         }

      else
         {
         cout << "No object has registered notification with SimClock "
              << "that it is interested in object:Population, "
              << "event:NT_TRANSFER";
         }

      }
      break;

   default:
      {
      cout << "Population dosen't know how to process that event";
      }

   } // end of SWITCH

return ES_UPDATE;   // keep generating widgets
}
```

```
Print date: 8/17/1992     Print time:  9:07
//-----------------------------------------------------------------
//  PROGRAM: wqueue.hpp
//
//  PURPOSE: A dynamic queue that holds Widget*'s
//
//  NOTE: Widget queue holds n number of widgets. This value can be
//        set to LONG_MAX to model unlimited holding capacity or
//        something less when queue capacity is a consideration.
//-----------------------------------------------------------------

#if !defined _WQUEUE_HPP
#define      _WQUEUE_HPP 1

//--------------------------- Includes --------------------------//

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif

#if !defined ( _TQUEUE_HPP )
#include <tqueue.hpp>
#endif

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

//----------------------- Forward references -------------------//

class Simclock;


class WidgetQueue : protected TQueue< Widget* >, public StatSimObj
   {
   protected:
      long queueCapacity;

      //-- get/put methods --//
      void PutWidget( void *pData );
      Widget *GetWidget( void );

   public:
      //-- constructor --//
      WidgetQueue( double _currentTime, long _queueCapacity );

      //-- notifications that this object expressed interest in --//
      virtual UINT Notify( SimEnv *pSimEnv, NOTIFY *pNotify );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  9:08
//-------------------------------------------------------------------
//  PROGRAM: wqueue.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _WQUEUE_HPP )
#include "wqueue.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include "stsimenv.hpp"
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

#if !defined ( _TQUEUE_HPP )
#include <tqueue.hpp>
#endif

#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif

//-- WidgetQueue::WidgetQueue() ---------------------------------
//
//-- constructor
//---------------------------------------------------------------

WidgetQueue::WidgetQueue( double _currentTime, long _queueCapacity )
   : TQueue< Widget* >(),
     StatSimObj( "WidQueue", 1, 0, NULL ),
     queueCapacity( _queueCapacity )
     {
     //-- statistics --//
     AddStatsObsObj( ID_TIME_IN_QUEUE_ALL, "Time all" );
     AddStatsObsObj( ID_TIME_IN_QUEUE_WAIT, "Time wait" );
     AddStatsTimeObj( ID_LENGTH_OF_QUEUE_ALL, "Len all", _currentTime );
     AddStatsTimeObj( ID_LENGTH_OF_QUEUE_WAIT, "Len wait", _currentTime );

     return;
     }


//-- WidgetQueue::Notify() --------------------------------------
//
//-- This object becomes the receiver of this notification message after
//   it registers interest in certain events from certain objects.
//---------------------------------------------------------------

UINT WidgetQueue::Notify( SimEnv *pSimEnv, NOTIFY *pNotify )
{
switch( pNotify->notifyType )
   {
   //-- receive object --//
   case NT_TRANSFER:
      {
      Widget *pWidget = ((Widget*) (*pNotify->pData));

      BOOL received = pSimEnv->Broadcast( (SimObj*) this,
            NT_TRANSFER, &(void*)pWidget );

      if( ! received )
         {
         //-- put in queue and update stats --//
         Put( pWidget );
         AddTimeObservation( ID_LENGTH_OF_QUEUE_WAIT, Length(),
               pSimEnv->GetCurrentTime() );
         AddTimeObservation( ID_LENGTH_OF_QUEUE_ALL, Length(),
               pSimEnv->GetCurrentTime() );

         }
      else
         {
         //-- update stats on zero time spent in queue --//
         AddObsObservation( ID_TIME_IN_QUEUE_ALL, 0 );
         AddTimeObservation( ID_LENGTH_OF_QUEUE_ALL, Length(),
               pSimEnv->GetCurrentTime() );

         }
      }
      break;

   //-- transfer object --//
   case NT_INIT_TRANSFER:
      {
      Widget *pWidget = Get();

      if( pWidget )
         {
         double timeInQueue = pSimEnv->GetCurrentTime() -
            pWidget->GetArrivalTime();

         AddObsObservation( ID_TIME_IN_QUEUE_WAIT, timeInQueue );
         AddObsObservation( ID_TIME_IN_QUEUE_ALL, timeInQueue );

         pNotify->pSender->Transfer( pSimEnv, (void*) pWidget );
         }

      }
      break;

   default:
      {
      cout << "WidgetQueue dosen't know how to process that notify type";
      return FALSE;
      }

   } // end of SWITCH
```

```
return TRUE;
}
```

```
Print date: 8/17/1992     Print time:  9:07
//-------------------------------------------------------------------
//  PROGRAM: rserver.hpp
//
//  PURPOSE: Provides n resources all of the same type. Similar to
//           servers in parallel.
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
   This object will new n resources. It will also register what
notification messages it will receive from certain simulation objects.
Resource servers collect statistics on group utilization, average busy
time per server, and average idle time per server.

   This object stores a list of resources and distributes work to idle
resources. There is circular communication between the resource server and
resources. Normally, when their are parallel resources, a resource
server is used. When resources are serial, or only one, then a resource
server is not needed. The resource server provides group statistics which
would not be available if n resources were acting independently.
---------------------------------------------------------------------
*/

#if !defined _RSERVER_HPP
#define      _RSERVER_HPP 1

//--------------------------- Includes --------------------------//

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _TDLIST_HPP )
#include <tdlist.hpp>
#endif


//------------------------- Forward reference -------------------//

class Resource;
class Widget;


template < class T >
class ResourceList : public TDList< T >
   {
   public:
      //-- constructor --//
      ResourceList( void )
         : TDList< Resource* > ()
         { }

      //-- get an idle resource --//
      T GetAvail( void );

      //-- get a count of the busy resources --//
      UINT GetBusyCount( void );

   };


class ResourceServer : public StatSimObj
   {
   protected:
      ResourceList < Resource* > resourceList;
      UINT resourcesScheduled;
      Resource *pIdleResource;

   public:
      //-- constructor --//
      ResourceServer( SimEnv *pSimEnv, UINT _resourcesScheduled,
            double serviceRate );

      //-- destructor --//
      ~ResourceServer( void );

      //-- notifications that this object expressed interest in --//
      virtual UINT Notify( SimEnv *pSimEnv, NOTIFY *pNotify );

      //-- widget transfer capability --//
      virtual UINT Transfer( SimEnv *pSimEnv, void *pWidget );

      //-- invoke a resource to start service --//
      void StartService( SimEnv *pSimEnv, Resource *pResource,
            Widget *pWidget );

      //-- discard widget object and request new widgets to serve --//
      void EndService( SimEnv *pSimEnv, Resource *pResource,
            Widget *pWidget );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:09
//---------------------------------------------------------------------
//  PROGRAM: rserver.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _RSERVER_HPP )
#include "rserver.hpp"
#endif

#if !defined ( _RESOURCE_HPP )
#include "resource.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

extern "C"
{
#include <stdio.h>
}

#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif


//------------------------- Forward references ----------------//

class WidgetQueue;


//-- ResourceList::GetAvail() ---------------------------------------
//
//-- gets the next available resource
//---------------------------------------------------------------------

template < class T >
T ResourceList< T >::GetAvail( void )
   {
   GoToHead();

   for( UINT i=0; i < length; i++ )
      {
      T pItem = Examine();

      if ( !pItem )    // empty list??
             return NULL;

      if ( !pItem->IsBusy() )
             return pItem;

      GoToNext();
      }

   return NULL;
   }


//-- ResourceList::GetBusyCount() -----------------------------------
//
//-- gets a count of the number of busy resources
//---------------------------------------------------------------------

template < class T >
UINT ResourceList< T >::GetBusyCount( void )
   {
   GoToHead();

   UINT busyCount = 0;

   for( UINT i=0; i < length; i++ )
      {
      T pItem = Examine();

      if ( !pItem )    // empty list??
             return NULL;

      if ( pItem->IsBusy() )
             busyCount++;

      GoToNext();
      }

   return busyCount;
   }


//-- ResourceServer::ResourceServer() -------------------------------
//
//-- constructor
//---------------------------------------------------------------------

ResourceServer::ResourceServer( SimEnv *pSimEnv,
      UINT _resourcesScheduled, double serviceRate )
   : StatSimObj( "ResServ", 1, 0, NULL ),
     resourceList(),
     resourcesScheduled( _resourcesScheduled ),
     pIdleResource( NULL )
   {
   char buffer1 [ 3 ];
   char buffer2 [ 12 ];
   long seed = 200000001;

   //-- resources available --//
   for( UINT i = 0; i < _resourcesScheduled; i++ )
      {
      seed = seed + 200000000;
```

```
      strcpy( buffer2, "Res " );
      sprintf( buffer1, "%u", i );
      strcat( buffer2, buffer1 );

      Resource *pResource = new Resource( pSimEnv->GetCurrentTime(),
            this, serviceRate );

      pResource->SetName( buffer2 );
      pResource->SetInitialSeed( seed );
      resourceList.Append( pResource );

      //-- register the WidgetQueue object with this SimEnv --//
      pSimEnv->RegisterSimObj( (SimObj*) pResource, RE_NOREGISTER );
      }

   //-- statistics --//
   AddStatsTimeObj( ID_CUMM_RESOURCE_UTIL, "Util", pSimEnv->GetCurrentTime() );
   AddStatsObsObj( ID_BUSY_NUMBER, "Busy" );
   AddStatsObsObj( ID_IDLE_NUMBER, "Idle" );

   return;
   }


//-- ResourceServer::~ResourceServer() ----------------------------
//
//-- destructor
//-----------------------------------------------------------------

ResourceServer::~ResourceServer( void )
   {
   resourceList.GoToHead();

   //-- release the resources --//
   for( UINT i = 0; i < resourceList.Length(); i++ )
      {
      Resource *pResource = resourceList.Examine();
      delete pResource;

      resourceList.GoToNext();
      }

   return;
   }


//-- ResourceServer::Notify() -------------------------------------
//
//-- This object becomes the receiver of this notification message after
//   it registers interest in certain events from certain objects.
//-----------------------------------------------------------------

UINT ResourceServer::Notify( SimEnv *pSimEnv, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      //-- receive object --//
      case NT_TRANSFER:
         {
         Resource *pResource = resourceList.GetAvail();
         if ( ! pResource )
            return FALSE;
         else
            StartService( pSimEnv, pResource, ((Widget*) (*pNotify->pData)) );
         }
         break;


      default:
         {
         cout << "ResourceServer dosen't know how to process that notify "
              << "type.";
         }

      } // end of switch

   return TRUE;
   }


//-- ResourceServer::Transfer() -----------------------------------
//
//-- A resource of this resource server is idle and wants to provide
//   service. The resource server will route an object to the resource
//   so that it may provide service.
//-----------------------------------------------------------------

UINT ResourceServer::Transfer( SimEnv *pSimEnv, void *pWidget )
   {
   StartService( pSimEnv, pIdleResource, (Widget*) pWidget );

   return TRUE;
   }


//-- ResourceServer::StartService() -------------------------------
//
//-- this object is no longer serving anything
//-----------------------------------------------------------------

void ResourceServer::StartService( SimEnv *pSimEnv,
      Resource *pResource, Widget *pWidget )
   {
   pResource->StartService( pSimEnv, pWidget );

   //-- collect stats --//
   UINT idleCount = resourceList.Length() - resourceList.GetBusyCount();
   AddObsObservation( ID_IDLE_NUMBER, (double) idleCount );

// new code to try to fix utilization stats !!!!!!!!!!!!!!!!
   double busyCount = (double) resourceList.GetBusyCount();
//    AddObsObservation( ID_BUSY_NUMBER, busyCount );
   AddTimeObservation( ID_CUMM_RESOURCE_UTIL,
                       ( busyCount / (double) resourcesScheduled ),
                       pSimEnv->GetCurrentTime() );
   AddObsObservation( ID_BUSY_NUMBER, busyCount );
// end new code here
```

```
   return;
   }


//-- ResourceServer::EndService() ---------------------------------
//
//-- this object is no longer serving anything
//-------------------------------------------------------------------

void ResourceServer::EndService( SimEnv *pSimEnv, Resource *pResource,
      Widget *pWidget )
   {
   //-- collect stats --//
   double busyCount = (double) resourceList.GetBusyCount();

// new code to try to fix utilization stats !!!!!!!!!!!!!!!!
//   busyCount++;
   UINT idleCount = resourceList.Length() - resourceList.GetBusyCount();
   AddObsObservation( ID_IDLE_NUMBER, (double) idleCount );
   AddObsObservation( ID_BUSY_NUMBER, busyCount );
// end new code here

/*
------
   AddTimeObservation( ID_CUMM_RESOURCE_UTIL,
                       ( busyCount / (double) resourcesScheduled ),
                       pSimEnv->GetCurrentTime() );
-------
*/

   //-- service complete, send on to next object --//
   UINT received = pSimEnv->Broadcast( (SimObj*) this, NT_TRANSFER,
         &(void*)pWidget );

   if ( ! received )
      delete pWidget;

   pIdleResource = pResource;

   pSimEnv->Broadcast( (SimObj*) this, NT_INIT_TRANSFER, NULL );

   return;
   }
```

```
Print date: 8/17/1992     Print time:  9:05
//-----------------------------------------------------------------
//  PROGRAM: resource.hpp
//
//  PURPOSE: Simulation objects of type resource
//
//  NOTE: Resources will let ResourceServers handle notification messages
//        if they were created by a ResourceServer or they will handle
//        their own notification messages if not.
//-----------------------------------------------------------------

#if !defined _RESOURCE_HPP
#define      _RESOURCE_HPP 1

//--------------------------- Includes --------------------------//

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

#if !defined ( _RANDOM_HPP )
#include <random.hpp>
#endif

//---------------------- Forward reference --------------------//
class Widget;
class RandExponential;
class ResourceServer;


class Resource : public StatSimObj
   {
   protected:
      //-- widget in service or NULL --//
      Widget *pIsServing;

      //-- present ResourceServer or NULL --//
      ResourceServer *pResourceServer;

      //-- service rate random variable --//
      Rand *pRand;

   public:
      //-- constructor --//
      Resource( double _currentTime, ResourceServer *_pResourceServer,
            double _serviceRate );

      //-- is resource busy --//
      Widget* IsBusy( void ) { return pIsServing; }

      //-- simulation update --//
      virtual int Update( SimEnv *pSimEnv, UINT eventType, void *pData );

      //-- notifications that this object expressed interest in --//
      virtual UINT Notify( SimEnv *pSimEnv, NOTIFY *pNotify );

      //-- widget transfer capability --//
      virtual UINT Transfer( SimEnv *pSimEnv, void *pWidget );

      //-- start serving a widget --//
      void StartService( SimEnv *pSimEnv, Widget *pWidget );

      //-- end serving a widget --//
      void EndService( SimEnv *pSimEnv, Widget *pWidget );

      void SetInitialSeed( long _seed ) { pRand->SetSeed( _seed ); }
   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:09
//-------------------------------------------------------------------
//  PROGRAM: resource.cpp

// resource statistics are not correct!  still?
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _RESOURCE_HPP )
#include "resource.hpp"
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

#if !defined ( _RSERVER_HPP )
#include "rserver.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include "stsimenv.hpp"
#endif

#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif



//-- Resource::Resource() ----------------------------------------
//
//-- constructors
//----------------------------------------------------------------

Resource::Resource( double _currentTime, ResourceServer *_pResourceServer,
      double _serviceRate )
   : StatSimObj( "Res", 1, 0, NULL ),
     pIsServing( NULL ),
     pResourceServer( _pResourceServer ),
     pRand( NULL )

      {
      //-- statistics --//
      AddStatsTimeObj( ID_RESOURCE_UTIL, "Util", _currentTime );

      //-- random number generator --//
      pRand = MakeRandExponential( _serviceRate, 0, "ServeRate" );
      }


//-- Resource::Update() ------------------------------------------
//
```

```
//-- update this simulation object
//----------------------------------------------------------------

int Resource::Update( SimEnv *pSimEnv, UINT eventType, void *pData )
   {
   switch( eventType )
      {
      case ET_DEPART:
         {
         Widget *pWidget = (Widget*) pData;

         EndService( pSimEnv, pWidget );
         }
         break;

      default:
         {
         cout << "Resources don't know how to process this event";
         }

      } // end of switch

   return ES_NOUPDATE; // no automatic reregistering
   }


//-- Resource::Notify() ------------------------------------------
//
//-- This object becomes the receiver of this notification message after
//   it registers interest in certain events from certain objects.
//----------------------------------------------------------------

UINT Resource::Notify( SimEnv *pSimEnv, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      //-- receive object --//
      case NT_TRANSFER:
         {
         if( pIsServing )
            return FALSE;
         else
            StartService( pSimEnv, ((Widget*) (*pNotify->pData)) );
         }
         break;

      default:
         {
         cout << "Resource dosen't know how to process that notify type.";
         }

      }  // end of switch

   return TRUE;
   }


//-- Resource::StartService() ------------------------------------
//
//-- schedule the end of service event for this object
```

```
//-------------------------------------------------------------

void Resource::StartService( SimEnv *pSimEnv, Widget *pWidget )
   {
   //-- mark the resource as busy --//
   pIsServing = pWidget;

   AddTimeObservation( ID_RESOURCE_UTIL, 0, pSimEnv->GetCurrentTime() );

   //-- get the service time --//
   double timeOfService = pRand->RandValue();

   pSimEnv->RegisterEvent( (SimObj*) this, timeOfService, 0,
         ET_DEPART, (void*) pWidget );

   return;
   }

//-- Resource::EndService() ----------------------------------
//
//-- this object is no longer serving anything
//------------------------------------------------------------

void Resource::EndService( SimEnv *pSimEnv, Widget *pWidget )
   {
   //-- mark the resource as idle --//
   pIsServing = NULL;

   AddTimeObservation( ID_RESOURCE_UTIL, 1, pSimEnv->GetCurrentTime() );

   if( pResourceServer )
      pResourceServer->EndService( pSimEnv, this, pWidget );
   else
      {
      UINT received = pSimEnv->Broadcast( (SimObj*) this, NT_TRANSFER,
            &(void*)pWidget );

      if( !received )
         delete pWidget;

      //-- pull a widget and provide service --//
      pSimEnv->Broadcast( (SimObj*) this, NT_INIT_TRANSFER, NULL );
      }

   return;
   }


//-- Resource::Transfer() ------------------------------------
//
//-- A resource is idle and wants to provide service.
//------------------------------------------------------------

UINT Resource::Transfer( SimEnv *pSimEnv, void *pWidget )
   {
   StartService( pSimEnv, (Widget*) pWidget );

   return TRUE;
   }
```

```
Print date: 8/17/1992     Print time:  9:05
//-----------------------------------------------------------------
//  PROGRAM: sink.hpp
//
//  PURPOSE: Simulation object of type sink
//
//  NOTE: Sink is a disposal point for the object arriving to the sink
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   This object will collect observational statistics on the widgets time
in the system. Widgets are always deleted in the sink.
-------------------------------------------------------------------
*/

#if !defined _SINK_HPP
#define      _SINK_HPP 1

//--------------------------- Includes -------------------------//

#if !defined ( _FLAGS_H )
#include "flags.h"
#endif

#if !defined ( _STATSIM_HPP )
#include <statsim.hpp>
#endif

class SimEnv;

class Sink : public StatSimObj
   {
   public:
      //-- constructor --//
      Sink( void );

      //-- notifications that this object expressed interest in --//
      virtual UINT Notify( SimEnv *pSimEnv, NOTIFY *pNotify );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  9:10
//-----------------------------------------------------------------
//  PROGRAM: sink.cpp
//-----------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//--------------------------- Includes -------------------------//

#if !defined ( _SINK_HPP )
#include "sink.hpp"
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _WIDGET_HPP )
#include "widget.hpp"
#endif

#if !defined ( __IOSTREAM_H )
extern "C"
{
#include <iostream.h>
}
#endif


//-- Sink::Sink() ---------------------------------------------
//
//-- constructor
//-------------------------------------------------------------

Sink::Sink( void )
   : StatSimObj( "Sink", 1, 0, NULL )
   {
   //-- statistics --//
   AddStatsObsObj( ID_TIME_IN_SYSTEM, "Time sys" );

   return;
   }


//-- Sink::Notify() -------------------------------------------
//
//-- This object becomes the receiver of this notification message after
//   it registers interest in certain events from certain objects
//-------------------------------------------------------------

UINT Sink::Notify( SimEnv *pSimEnv, NOTIFY *pNotify )
   {
   switch( pNotify->notifyType )
      {
      case NT_TRANSFER:
         {
         Widget *pWidget = ((Widget*) (*pNotify->pData));

         if( pWidget )
            {
            double timeInSystem = pSimEnv->GetCurrentTime() -
               pWidget->GetArrivalTime();

            //-- something wants to come to the sink --//
            AddObsObservation( ID_TIME_IN_SYSTEM, timeInSystem );

            delete pWidget;
            }
         }
         break;

      default:
         {
         cout << "Sinks don't know how to handle that message";
         return FALSE;
         }

      } // end of switch

return TRUE;
}
```

Appendix F: Abstract classes SimObject & StatSimObject

```
Print date: 9/29/1992     Print time: 16:04
//---------------------------------------------------------------------
//  PROGRAM: SIMOBJ.H
//---------------------------------------------------------------------


#if !defined _SIMOBJ_H
#define      _SIMOBJ_H  1

#if !defined ( _TARRAY_HPP )   // for trace array
#include <tarray.hpp>
#endif

#if !defined ( _TYPEDEFS_HPP )
#include <typedefs.hpp>
#endif

const int SO_MAXCAPTIONLENGTH = 32;


struct TRACE_DATA
   {
   char *name;
   TYPE typeFlag;
   void *pData;

   TRACE_DATA( void ) : name( NULL ), typeFlag( TYPE_INT ), pData( NULL ) { }

   TRACE_DATA( TRACE_DATA &td )
      : name     ( NULL ),
        typeFlag( td.typeFlag ),
        pData    ( td.pData )
      {
      if ( td.name )
         { name = new char[ strlen( td.name ) + 1 ]; strcpy( name, td.name ); }
      }


   ~TRACE_DATA( void ) { if ( name ) delete name; }
   };



#endif
```

```
Print date: 8/17/1992     Print time:  8:55
//--------------------------------------------------------------------
// PROGRAM: SIMOBJ.HPP
//
// PURPOSE: provides definition for generic simulation objects
//
// NOTE: This is an abstract class
//--------------------------------------------------------------------

#if !defined _SIMOBJ_HPP
#define      _SIMOBJ_HPP   1

//------------------------- Includes ----------------------------//

#if !defined ( _SIMENV_H )
#include <simenv.h>
#endif

#if !defined ( __STRING_H )
#include <string.h>
#endif

#if !defined ( _TYPEDEFS_HPP )
#include <typedefs.hpp>
#endif

#if !defined ( _SIMOBJ_H )
#include <simobj.h>
#endif


//---------------------- Forward Declarations--------------------//

class GDataObj;
class SimEnv;
class StatSimEnv;

struct NOTIFY;


//----------------------- SimObj Class ---------------------------//

class SimObj
   {
   friend class SimEnv;
   friend class StatSimEnv;
   friend BOOL IntRKS( SimObj *pSimObj, double timeStep, double currentTime,
                       int svCount, double *stateVar, STATEPROC lpFn );
   protected:
      //-- statics --//
      static UINT nextHandle;   // global handle incrementer
      //SimEnv *pSimEnv;          // pointer to the currently executing SimEnv

      //-- member data --//
      UINT        handle;       // unique handle for this object

      char       *name;
      double      timeStep;
      int         priority;
      BOOL        deleteOnDelete; // delete when simEnv deleted?

      GDataObj   *pSimData;     // pointer to a data container object
      TraceArray *pTraceArray;  // non-NULL if we are tracing this object

      BOOL        notify;       // TRUE if another object expresses interest
                                // in this object

      virtual void State( int svCount, double *stateVar,
            double *derive, double time ) { }

      //-- constructors --//
      SimObj( void );
      SimObj( char *_name, double _timeStep, int _priority, GDataObj *_pData

      SimObj( SimObj& );    // copy constructor

   public:
      virtual ~SimObj( void );

   protected:
      //--- access data members ---//
      void SetTimeStep( double _timeStep ) { timeStep = _timeStep;  }
      void SetPriority( int    _priority ) { priority = _priority;  }
      void SetDataPtr ( GDataObj *_pData ) { pSimData = _pData;     }

   public:
      //-- Trace inspector --//
      //-- start/stop tracing --//
      virtual void StartTrace( void ) { }
      void         StopTrace( void )
                 { if ( pTraceArray ) delete pTraceArray;  pTraceArray = NULL;

      TraceArray *InspectProperty( void ) { return pTraceArray; }

   protected:
      //-- ( virtual ) Update routine --//
      virtual int Update( SimEnv *pSimEnv, UINT eventType, void *_pData )
           { return ES_NOUPDATE; }

      //-- ( virtual ) will use default if children don't override --//
      virtual UINT Notify( SimEnv *pSimEnv, NOTIFY *pNotify )
            { return FALSE; }

      virtual LONG IsDistsOn          ( void ) { return (LONG) NULL; }
      virtual LONG IsObsStatsOn       ( void ) { return (LONG) NULL; }
      virtual LONG IsContinStatsOn    ( void ) { return (LONG) NULL; }
      virtual LONG IsTimeStatsOn      ( void ) { return (LONG) NULL; }
      virtual LONG IsCountStatsOn     ( void ) { return (LONG) NULL; }

   public:
      double    GetTimeStep( void ) { return timeStep; }
      int       GetPriority( void ) { return priority; }
      const char* const GetName( void ) { return (const char* const) name; }
      void      SetName    ( char *_name );
      GDataObj *GetDataPtr ( void ) { return pSimData; }

      UINT      GetHandle( void ) { return handle; }

      //-- transfer a widget --//
      virtual UINT Transfer( SimEnv *pSimEnv, void *pWidget ) { return FALSE;
```

```
      virtual void ReInitializeAllStats( double _currentTime ) { return; }
   };


inline
SimObj::~SimObj( void )
   {
   if ( pTraceArray )
      delete pTraceArray;

   if ( name )
      delete name;
   }


inline
void SimObj::SetName( char  *_name )
   {
   if ( name )
      delete name;

   if ( _name )
      {
      name = new char[ strlen( _name ) + 1 ];
      strcpy( name, _name );
      }
   else
      name = NULL;
   }

#endif
```

```
Print date: 8/17/1992     Print time:  8:56
//---------------------------------------------------------------
//-- SIMOBJ.CPP
//---------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

#include <simobj.hpp>

#include <dataobj.hpp>


//--------------------------- Statics --------------------------//

UINT SimObj::nextHandle = 0;


//-- SimObj::SimObj() ------------------------------------------
//
//-- constructors
//---------------------------------------------------------------

SimObj::SimObj( void )
   : timeStep      ( 1 ),
     priority      ( 0 ),
     deleteOnDelete( FALSE ),
     pSimData      ( NULL  ),
     pTraceArray   ( NULL  ),
     notify        ( TRUE  ),
     name          ( NULL  )
   {
   handle = nextHandle++;
   }


SimObj::SimObj( char *_name, double _timeStep, int _priority, GDataObj *_pData )
   : timeStep      ( _timeStep ),
     priority      ( _priority ),
     deleteOnDelete( FALSE ),
     pSimData      ( _pData ),
     pTraceArray   ( NULL ),
     notify        ( TRUE ),
     name          ( NULL )
   {
   SetName( _name );
   handle = nextHandle++;
   }


//-- SimObj::SimObj() ------------------------------------------
//
//-- constructors
//---------------------------------------------------------------

SimObj::SimObj( SimObj &simObj )
   : timeStep   ( simObj.timeStep ),
     priority   ( simObj.priority ),
     deleteOnDelete( simObj.deleteOnDelete ),
     pSimData   ( NULL ),
     pTraceArray( NULL ),
     notify     ( simObj.notify ),
     name       ( NULL )
{
handle = nextHandle++;

if ( simObj.pSimData )
   pSimData = new GDataObj( *(simObj.pSimData) );   // copy constructor av
else
   pSimData = NULL;

if ( simObj.pTraceArray )
   pTraceArray = new TraceArray( *(simObj.pTraceArray) );   // copy constr
else
   pTraceArray = NULL;

SetName( simObj.name );
}
```

```
Print date: 8/17/1992     Print time:  8:56
//-----------------------------------------------------------------
// PROGRAM: STATSIM.HPP
//
// PURPOSE: provides definition for generic simulation objects
//          with statistics
//
// NOTE: This is an abstract class
//-----------------------------------------------------------------

#if !defined _STATSIM_HPP
#define      _STATSIM_HPP   1

//------------------------- Includes ----------------------------//

#if !defined ( _SIMOBJ_HPP )
#include <simobj.hpp>
#endif

#if !defined ( _ST_ARRAY_HPP )
#include <st_array.hpp>
#endif

#if !defined ( _RANDOM_HPP )
#include <random.hpp>
#endif

//-------------------- StatSimObj Class ------------------------//

class StatSimObj : public SimObj
   {
   private:
      //-- statistics (NULL if not in use) --//
      DistArray           *pDistArray;
      StatsContinArray    *pStatsContinArray;
      StatsTimeArray      *pStatsTimeArray;
      StatsObsArray       *pStatsObsArray;
      CounterArray        *pCounterArray;

      UINT AddDist( Rand *pRand );

   protected:
      //-- add statistics objects --//
      Rand *MakeRandUniform( double lowerBound, double upperBound,
            long seed = 200000001, char *_name = "-" );

      Rand *MakeRandExponential( double mean,
            long seed = 200000001, char *_name = "-" );
/*
----------------------------
      Rand *MakeRandTriangular( double _min, double _mode, double _max,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandWeibull( double shape, double scale,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandPoisson( double mean,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandErlang( double shape, double scale,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandBernoulli( double probability,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandNormal( double mean, double stdev,
            UINT stream = 1, long seed = 200000001 );

      Rand *MakeRandLogNormal( double mean, double stdev,
            UINT stream = 1, long seed = 200000001 );
-------------------------------
*/

      //-- gather statistics --//
      UINT AddStatsContinObj ( UINT id, char *_name, double currentTime,
            double initialValue );
      UINT AddStatsTimeObj ( UINT id, char *_name, double currentTime );
      UINT AddStatsObsObj  ( UINT id, char *_name );
      UINT AddStatsCountObj( UINT id, char *_name );

      BOOL AddContinObservation ( UINT id, double value, double currentTime )
      BOOL AddTimeObservation ( UINT id, double value, double currentTime );
      BOOL AddObsObservation  ( UINT id, double value );
      BOOL AddCountObservation( UINT id, UINT value   );

   public:
      //-- constructors --//
      StatSimObj( void )
         : SimObj(),
           pDistArray   ( NULL ),
           pStatsTimeArray      ( NULL ),
           pStatsContinArray( NULL ),
           pStatsObsArray       ( NULL ),
           pCounterArray        ( NULL )
         { }

      StatSimObj( char *_name, double _timeStep, int _priority,\
         GDataObj *_pData )
         : SimObj( _name, _timeStep, _priority, _pData ),
           pDistArray   ( NULL ),
           pStatsTimeArray      ( NULL ),
           pStatsContinArray( NULL ),
           pStatsObsArray       ( NULL ),
           pCounterArray        ( NULL )
         { }

      ~StatSimObj( void );

      //-- ( virtual ) Update routine --//
      virtual int Update( SimEnv *pSimEnv, UINT type, void *pData )
            { return ES_NOUPDATE; }

      virtual LONG IsDistsOn( void )
                           { return (LONG) pDistArray; }

      virtual LONG IsObsStatsOn( void )
                           { return (LONG) pStatsObsArray; }

      virtual LONG IsContinStatsOn( void )
                           { return (LONG) pStatsContinArray; }
```

```
      virtual LONG IsTimeStatsOn( void )
                        { return (LONG) pStatsTimeArray; }

      virtual LONG IsCountStatsOn( void )
                        { return (LONG) pCounterArray; }


      //-- a child should not override this --//
      virtual void ReInitializeAllStats( double _currentTime );


   };


inline
Rand *StatSimObj::MakeRandUniform( double lowerBound, double upperBound,
      long seed, char *_name )
   {
   Rand *pRand = (Rand*) new RandUniform( lowerBound, upperBound,
         seed, _name );

   AddDist( pRand );

   return pRand;
   }

inline
Rand *StatSimObj::MakeRandExponential( double mean, long seed, char *_name )
   {
   Rand *pRand = (Rand*) new RandExponential( mean, seed, _name );

   AddDist( pRand );

   return pRand;
   }


inline
UINT StatSimObj::AddDist( Rand *pRand )
   {
   if ( ! pDistArray )
      pDistArray = new DistArray;

   return pDistArray->Append( pRand );
   }



#endif
```

```
Print date: 8/17/1992     Print time:  8:56
//---------------------------------------------------------------------
//-- STATSIM.CPP
//
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

#include <statsim.hpp>

#include <st_array.hpp>


//--------------------- CounterArray methods ---------------------//
Counter *CounterArray::GetCounter( UINT id )
   {
   for ( UINT i=0; i < length; i++ )
      if ( array[ i ]->GetID() == id )
         return array[ i ];

   return NULL;
   }


BOOL CounterArray::RecordValue( UINT id, long value )
   {
   Counter *pCounter = GetCounter( id );

   if ( ! pCounter )
      return FALSE;
   else
      {
      pCounter->RecordValue( value );
      return TRUE;
      }
   }


//---------------------- StatArray methods ----------------------//

StatsContin *StatsContinArray::GetStatObj( UINT id )
   {
   for ( UINT i=0; i < length; i++ )
      if ( array[ i ]->GetID() == id )
         return array[ i ];

   return NULL;
   }


StatsTime *StatsTimeArray::GetStatObj( UINT id )
   {
   for ( UINT i=0; i < length; i++ )
      if ( array[ i ]->GetID() == id )
         return array[ i ];

   return NULL;
   }


StatsObs *StatsObsArray::GetStatObj( UINT id )
   {
   for ( UINT i=0; i < length; i++ )
      if ( array[ i ]->GetID() == id )
         return array[ i ];

   return NULL;
   }

BOOL StatsContinArray::RecordValue( UINT id, double value, double currentTime
   {
   StatsContin *pStats = GetStatObj( id );

   if ( ! pStats )
      return FALSE;
   else
      {
      pStats->RecordValue( value, currentTime );
      return TRUE;
      }
   }


BOOL StatsTimeArray::RecordValue( UINT id, double value, double currentTime )
   {
   StatsTime *pStats = GetStatObj( id );

   if ( ! pStats )
      return FALSE;
   else
      {
      pStats->RecordValue( value, currentTime );
      return TRUE;
      }
   }


BOOL StatsObsArray::RecordValue( UINT id, double value )
   {
   StatsObs *pStats = GetStatObj( id );

   if ( ! pStats )
      return FALSE;
   else
      {
      pStats->RecordValue( value );
      return TRUE;
      }
   }

//-- constructors --//

StatSimObj::~StatSimObj( void )
   {
   if ( pStatsTimeArray )
      delete pStatsTimeArray;
```

```
    if ( pStatsContinArray )
       delete pStatsContinArray;

    if ( pStatsObsArray )
       delete pStatsObsArray;

    if ( pCounterArray )
       delete pCounterArray;
    }


UINT StatSimObj::AddStatsContinObj( UINT id, char *_name, double currentTime,
       double initialValue )
    {
    //-- need to make the array? --//
    if ( ! pStatsContinArray )
       pStatsContinArray = new StatsContinArray;

    // make a stats object
    StatsContin *pStatsContin =
          new StatsContin( id, _name, currentTime, initialValue );


    // add to the stats array
    return pStatsContinArray->Append( pStatsContin );
    }


UINT StatSimObj::AddStatsTimeObj( UINT id, char *_name, double currentTime )
    {
    //-- need to make the array? --//
    if ( ! pStatsTimeArray )
       pStatsTimeArray = new StatsTimeArray;

    // make a stats object
    StatsTime *pStatsTime = new StatsTime( id, _name, currentTime );

    // add to the stats array
    return pStatsTimeArray->Append( pStatsTime );
    }


UINT StatSimObj::AddStatsObsObj( UINT id, char *_name )
    {
    //-- need to make the array? --//
    if ( ! pStatsObsArray )
       pStatsObsArray = new StatsObsArray;

    //-- make a stats object
    StatsObs *pStatsObs = new StatsObs( id, _name );

    //-- add to the stats array
    return pStatsObsArray->Append( pStatsObs );
    }


UINT StatSimObj::AddStatsCountObj( UINT id, char *_name )
    {
    //-- need to make the array? --//
    if ( ! pCounterArray )
       pCounterArray = new CounterArray;

    // make a stats object
    Counter *pCounter = new Counter( id, _name );

    // add to the stats array
    return pCounterArray->Append( pCounter );
    }


BOOL StatSimObj::AddContinObservation( UINT id, double value, double currentT
    {
    if ( pStatsContinArray )
       return pStatsContinArray->RecordValue( id, value, currentTime );
    else
       return FALSE;
    }


BOOL StatSimObj::AddTimeObservation( UINT id, double value, double currentTin
    {
    if ( pStatsTimeArray )
       return pStatsTimeArray->RecordValue( id, value, currentTime );
    else
       return FALSE;
    }


BOOL StatSimObj::AddObsObservation( UINT id, double value )
    {
    if ( pStatsObsArray )
       return pStatsObsArray->RecordValue( id, value );
    else
       return FALSE;
    }


BOOL StatSimObj::AddCountObservation( UINT id, UINT value )
    {
    if ( pCounterArray )
       return pCounterArray->RecordValue( id, value );
    else
       return FALSE;
    }


void StatSimObj::ReInitializeAllStats( double _currentTime )
    {
    UINT length = 0;

    if ( pDistArray )
       {
       length = pDistArray->Length();
       for ( UINT i = 0; i < length; i++ )
          {
          Rand *pRand = pDistArray->Get( i );
          pRand->ReInitializeStats();
          }
       }
```

```
if ( pStatsContinArray )
   {
   length = pStatsContinArray->Length();
   for ( UINT i = 0; i < length; i++ )
      {
      StatsContin *pStatsContin = pStatsContinArray->Get( i );
      pStatsContin->ReInitializeStats( _currentTime );
      }
   }

if ( pStatsTimeArray )
   {
   length = pStatsTimeArray->Length();
   for ( UINT i = 0; i < length; i++ )
      {
      StatsTime *pStatsTime = pStatsTimeArray->Get( i );
      pStatsTime->ReInitializeStats( _currentTime );
      }
   }

if ( pStatsObsArray )
   {
   length = pStatsObsArray->Length();
   for ( UINT i = 0; i < length; i++ )
      {
      StatsObs *pStatsObs = pStatsObsArray->Get( i );
      pStatsObs->ReInitializeStats( _currentTime );
      }
   }

if ( pCounterArray )
   {
   length = pCounterArray->Length();
   for ( UINT i = 0; i < length; i++ )
      {
      Counter *pCounter = pCounterArray->Get( i );
      pCounter->ReInitializeStats( );
      }
   }

return;
}
```

Appendix G: Simulation environment classes SimEnv & StatSimEnv

```
Print date: 9/29/1992     Print time: 13:59
//---------------------------------------------------------------------
//  PROGRAM: SIMENV.H
//---------------------------------------------------------------------

#if !defined  _SIMENV_H
#define       _SIMENV_H 1


#if !defined ( __STRING_H )
#include <string.h>
#endif

//------------------------- Constants ----------------------------//

//-- simenv Run() return flags --//
const int SC_NOEVENTS       = -1;   // no events on the event list
const int SC_EOS            =  0;   // end of simulation (sysStopTime reached)
const int SC_TERMINATED     =  1;   // terminated by user event


//-- event status return flags ( Update() return values ) --//
const int ES_ERROR          = -1;
const int ES_UPDATE         =  0;
const int ES_NOUPDATE       =  1;
const int ES_TERMINATE      =  2;

//-- event registration flags --//
const int RE_NOALLOC        =  1;
const int RE_NOSIMOBJ       =  2;

const int RE_REGISTER       = 10;
const int RE_NOREGISTER     = 11;

const int MAXCAPTIONLENGTH = 25;
const int OUTPUTBUFFER     = 128;

enum EVENT
   {
   ET_DEFAULT = 32000,
   ET_STARTTRACE,             // start a trace on this simulation object
   ET_STOPTRACE,              // end a trace on this simulation object
   ET_STARTEVENTLIST,         // start viewing the event list each update
   ET_STOPEVENTLIST,          // stop viewing the event list each update
   ET_ENUMEVENTLIST,          // enum the event list at current event time
   ET_CLEARSTATS,             // clear all statistics
   ET_ENDSIMULATION           // terminate the simulation
   };


struct REPORT
   {
   char  simObjName[13];
   char  idName[13];
   char  distName[13];
   char  paramA[13];
   char  paramB[13];
   char  paramC[13];
   long  initialSeed;
   long  multiplier;
   float average;
   float sDeviation;
   float variation;
   float minimum;
   float maximum;
   long  observations;
   float halfWidth;
   float LCL;
   float UCL;
   float finalValue;
   long  count;

   //-- constructor --//
   REPORT( void )
      : initialSeed( 0 ),
        multiplier( 0 ),
        average( -1 ),
        sDeviation( -1 ),
        variation( -1 ),
        minimum( -1 ),
        maximum( -1 ),
        observations( 0 ),
        halfWidth( -1 ),
        LCL( -1 ),
        UCL( -1 ),
        finalValue( -1 ),
        count( 0 )

      {
      strcpy( simObjName, "-" );
      strcpy( idName, "-" );
      strcpy( distName, "-" );
      strcpy( paramA, "-" );
      strcpy( paramB, "-" );
      strcpy( paramC, "-" );
      }

   };

//---------------------- Forward references ---------------------//

class SimObj;
class SimEnv;
class TraceArray;

//------------------------ Typedefs -----------------------------//

//-- SimObject State Variable method --//
typedef void (SimObj::*STATEPROC)( int, double*, double*, double );

//-- application-defined trace function --//
typedef int (*TRACEPROC)( SimEnv*, SimObj*, UINT, TraceArray* );

//-- application-defined view event function --//
typedef void (*EVENTPROC)( SimObj* pSimObj, double eventTime,
                           UINT priority, UINT eventType );

//-- user defined method to replicate for each run of the simulation --//
typedef void (*REPPROC)( SimEnv* );
```

```
typedef int (*REPORTPROC)( REPORT& );
#endif
```

```
Print date: 8/17/1992     Print time:  8:40
//-------------------------------------------------------------------
//  PROGRAM: SIMENV.HPP
//
//  PURPOSE: Class declaration for the Simulation Environment object
//
//  COLLABORATORS: SimObj, TDList
//-------------------------------------------------------------------

#if !defined  _SIMENV_HPP
#define       _SIMENV_HPP 1

//------------------------- Includes ----------------------------//

#if !defined ( _SIMENV_H )
#include <simenv.h>
#endif

#if !defined ( _SYSSIMOB_HPP )
#include <syssimob.hpp>
#endif

#if !defined ( _TDLIST_HPP )
#include <tdlist.hpp>
#endif

#if !defined ( _TPARRAY_HPP )
#include <tparray.hpp>
#endif

//-- if defined at compile time --//
#if ! defined ( _Windows )
extern "C"
{
#include <iostream.h>
#include <conio.h>
}
#endif


//------------------------- Constants --------------------------//

//-- event tick timing sort flags --//
const int ET_TICK_FOUND_GT  =  0;
const int ET_TICK_FOUND_EQ  =  1;
const int ET_EOL            =  2;

//-- event priority sort flags --//
const int EP_EOL            =  0;
const int EP_PRIORITY_FOUND =  1;

//-- simobj search flags --//
const int SC_SIMOBJ_NOT_FOUND = 0;
const int SC_SIMOBJ_FOUND     = 1;


class SimObjArray : public TPtrArray< SimObj* >
   {
   public:
      SimObjArray( void ) { deleteOnDelete = FALSE; }
   };

struct NOTIFYFILTER
   {
   UINT notifyType;
   SimObjArray *pSimObjFilter;  // SimObjs receiving events for which
                                // notification messages should be sent
   };


class NotifyFilterArray : public TPtrArray< NOTIFYFILTER* >
   {
   public:
      UINT FindFilter( UINT notifyType );
   };


struct SIMOBJ
   {
   SimObj *pSimObj;             // simulation object address
   //long    tickIncr;           // simulation tick increment
   //int     priority;           // simulation priority
   NotifyFilterArray *pNotifyFilter;

   ~SIMOBJ( void ) { if ( pNotifyFilter ) delete pNotifyFilter; }
   };


struct SIMEVENT
   {
   SIMOBJ *pSimObjStruct;       // simulation object struct address
   long    eventTick;           // time when event occurs
   UINT    eventType;           // SimObj defined event type
   void   *pData;               // anything extra
   };


class SIMOBJList : public TDList< SIMOBJ* >
   {
   public:
      SIMOBJ *Find( SimObj *pSimObj );
   };


class SIMEVENTList : public TDList< SIMEVENT* >
   {
   };


struct NOTIFY

   {
   SimObj *pSender;             // object that posted the notification message
   UINT    notifyType;          // SimObj defined notify type
   void  **pData;               // one or two way transfer of info
   };


class SimEnv
```

```
{
protected:
   //-- simulation name --//
   char  *name;

   //-- timer variables --//
   double sysStartTime;       // starting time for simulation
   double sysCurrentTime;     // current time
   double sysTimeStep;        // system time step
   double sysStopTime;        // end of simulation

   long   sysCurrentTick;   // current tick count
   double sysCumTime;       // cumulative time (between successive runs)

   SIMOBJList   simObjList;      // list of simulation objects
   SIMEVENTList simEventList;    // list of simulation events

   //-- clock simulation object --//
   static SysSimObj sysSimObj;
   SIMOBJ *RegisterSystemSimObj( void );

   //-- protected methods --//
   int  SortEventTick( long nextTick );
   int  SortEventPriority( long eventTick, int priority );

   //-- statistic flags --//
   UINT distStats;
   UINT obsStats;
   UINT timeStats;
   UINT continStats;
   UINT countStats;
   void SetStatisticFlags( void );

   //-- run pre and post processing --//
   virtual UINT StartRun( void );
   virtual UINT EndRun( UINT terminateFlag );

   //-- reset state/statistics --//
   double lastClearStatsTime; // last time statistics were cleared
   virtual void ClearStatisticFlags( void );

   void GetSimObjName( SimObj *pSimObj, REPORT &report );

   BOOL deleteSimObjs;  // flag indicating that SimObjs should be deleted
                        // on deletion of the SimEnv;

   TRACEPROC  traceProc;  // application-defined trace function

   void (*localReportProc )( SimEnv* );


public:
   //-- constructors --//
   SimEnv( char *_name );

   SimEnv( char *_name, double startTime, double stopTime,\
           double timeStep, SIMOBJList simObjects );

   //-- destructor --//
   ~SimEnv( void );

   //-- set clock control variables --//
   void SetStartTime( double startTime ) { sysStartTime = startTime; }
   void SetTimeStep ( double timeStep  ) { sysTimeStep = timeStep; }
   void SetStopTime ( double stopTime  ) { sysStopTime = stopTime; };
   void ResetCumTime( void ) { sysCumTime = 0.0; };
   void ClearSystemState( void );


   //-- get clock control variables --//
   double GetStartTime  ( void )       { return sysStartTime;   }
   double GetTimeStep   ( void )       { return sysTimeStep;    }
   double GetStopTime   ( void )       { return sysStopTime;    }
   double GetCumTime    ( void )       { return sysCumTime;     }
   double GetCurrentTime( void )       { return sysCurrentTime; }

   //-- statistics info --//
   void GetStatisticLengths( UINT *pDistLength, UINT *pObsLength,
         UINT *pTimeLength, UINT *pContinLength,
         UINT *pCountLength );

   //-- get access to the simulation event list --//
   TDList <SIMEVENT*> *GetEventListPtr( void ) { return &simEventList;  }

   //-- maintain simulation object list --//
   BOOL RegisterSimObj ( SimObj *pSimObj, int registerFlag=RE_REGISTER );
   BOOL DeleteSimObj   ( SimObj *pSimObj );
   void ClearSimObjList( void ) { simObjList.Empty(); }
   void SetDeleteFlag( BOOL flag ) { deleteSimObjs = flag; }

   BOOL AddNotifyFilter( SimObj *pObjectToNotify, UINT notifyType, \
         SimObjArray *pBroadcastingObjects );

   BOOL AddNotifyFilter( SimObj *pObjectToNotify, UINT notifyType,\
         SimObj *pBroadcastingObject );


   //-- maintain event list --//
   BOOL RegisterEvent( SimObj* pSimObj, double eventIncr, int priority, \
                       UINT eventType, void *pData );
   BOOL RegisterEventStruct( SIMEVENT *pSimEventStruct, int priority );
   BOOL DeleteEvent( SIMOBJ *pSimObjStruct );
   void ClearEventList( void ) { simEventList.Empty(); }

   void EnumEvents( EVENTPROC );

   //-- notify objects --//
   BOOL  Broadcast( SimObj *pSender, UINT notifyType, void **pData );
//    void* Broadcast( SimObj *pSender, UINT notifyType )
//          { return NULL; }  // temporary code

   //--- control simulation ---//
   int  Run( void );

   void StopSimulation( void );

   //-- enumerate observational statistics to a user defined function --//
   BOOL DistStatsAvailable  ( void )  { return (BOOL) distStats;   }
   BOOL ObsStatsAvailable   ( void )  { return (BOOL) obsStats;    }
   BOOL TimeStatsAvailable  ( void )  { return (BOOL) timeStats;   }
```

```
    BOOL ContinStatsAvailable ( void )  { return (BOOL) continStats;  }
    BOOL CountStatsAvailable( void )  { return (BOOL) countStats; }
    virtual BOOL GlobalStatsAvailable( void ) { return FALSE; }

    void EnumDistStats              ( REPORTPROC );
    void EnumObsStats               ( int (*lpReportFn)( REPORT& ) );
    void EnumTimeStats              ( int (*lpReportFn)( REPORT& ) );
    void EnumContinStats            ( int (*lpReportFn)( REPORT& ) );
    void EnumCountStats             ( int (*lpReportFn)( REPORT& ) );

    //-- Debug features --//
    void SetTraceProc( TRACEPROC _traceProc ) { traceProc = _traceProc; }

    void SetReportProc( void (*reportProc)(SimEnv*) )
                    { localReportProc = reportProc; }

    void StartTracingAllObjects( void );
    void StopTracingAllObjects( void );


    double GetLastClearStatsTime( void ) { return lastClearStatsTime; }
    void ReInitializeAllStats( double _currentTime );

    void SetName( char * );
    const char* const GetName( void ) { return (const char* const) name; }

};


//---------------------------- Inlines --------------------------//

inline
BOOL SimEnv::AddNotifyFilter( SimObj *pObjectToNotify, UINT notifyType,\
     SimObj *pBroadcastingObject )
  {
  SimObjArray simObjFilter;
  simObjFilter.Append( (SimObj*) pBroadcastingObject );

  BOOL result = AddNotifyFilter( pObjectToNotify, notifyType, &simObjFilter );

  return result;
  }


inline
void SimEnv::SetName( char  *_name )
  {
  if ( name )
    delete name;

  if ( _name )
    {
    name = new char[ strlen( _name ) + 1 ];
    strcpy( name, _name );
    }
  else
    name = NULL;
  }
```

```
//---------------------- Integration Methods --------------------//

//-- public Runge-kutta integrator (uses virtual SimObj::State() )--//
BOOL IntRKS( SimObj *pSimObj,
             double  timeStep,
             double  currentTime,
             int     svCount,
             double *stateVar,
             STATEPROC lpFn = NULL);


#endif
```

```
Print date: 8/17/1992     Print time:  8:40
//-------------------------------------------------------------------
//  PROGRAM: SimEnv.cpp
//-------------------------------------------------------------------

#include <windows.h>  // for eventListWindow messaging only!!
#pragma hdrstop

//------------------------ Includes ----------------------------//

#include <tdlist.hpp>
#include <simobj.hpp>
#include <simenv.hpp>

#include <st_array.hpp>

#include <typedefs.hpp>

#include <limits.h>


#if !defined ( __STDIO_H )
#include <stdio.h>
#endif


//------------------------ Static data -------------------------//

SysSimObj SimEnv::sysSimObj;     // system simulation object


//------------------------ Externals ---------------------------//

extern HWND hWndEvent;


//------------------------ C Functions -------------------------//

void YieldControl( void );


/*
---------------------------------------------------------------------
   The SimEnv ( simulation clock ) is designed for discrete or
   continuous simulations.
---------------------------------------------------------------------
*/



//-- SimEnv::SimEnv() ----------------------------------------
//
//-- constructor methods
//------------------------------------------------------------------

//--- No arguments (simObjList empty) ---//
SimEnv::SimEnv( char *_name )
   : name           ( NULL ),
     distStats      ( FALSE ),  // initially, no observational statistics
     obsStats       ( FALSE ),
     continStats    ( FALSE ),
     timeStats      ( FALSE ),
     countStats     ( FALSE ),
     lastClearStatsTime ( 0 ),
     deleteSimObjs  ( FALSE ),
     traceProc      ( NULL  ),
     localReportProc( NULL  )
   {
   sysStartTime = sysCurrentTime = sysCumTime = 0.0;
   sysTimeStep = 1.0e-6;
   sysStopTime  = 1;
   sysCurrentTick = 0;

   SetName( _name );
   }


SimEnv::SimEnv( char *_name, double startTime, double stopTime,\
                double timeStep, SIMOBJList simObjects )
   : name           ( NULL ),
     distStats      ( FALSE ),   // initially, no observational statistics
     obsStats       ( FALSE ),
     continStats    ( FALSE ),
     timeStats      ( FALSE ),
     countStats     ( FALSE ),
     lastClearStatsTime ( 0 ),
     deleteSimObjs  ( FALSE ),
     traceProc      ( NULL  ),
     localReportProc( NULL  )
   {
   sysStartTime   = sysCurrentTime = sysCumTime = startTime;
   sysTimeStep    = timeStep;
   sysStopTime    = stopTime;
   simObjList     = simObjects;
   sysCurrentTick = 0;

   SetName( _name );
   }


//-- SimEnv::~SimEnv() ---------------------------------------
//
//-- destructor - deletes all registered simobjects if deleteSimObjs
//                flag is set
//------------------------------------------------------------------

SimEnv::~SimEnv( void )
   {
   if ( name )
      delete name;

   //-- clean up simObjList --//
   simObjList.GoToHead();

   for ( UINT node=0; node < simObjList.Length(); node++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();

      if ( deleteSimObjs || pSimObjStruct->pSimObj->deleteOnDelete )
```

```
         delete pSimObjStruct->pSimObj;

      delete pSimObjStruct;
      }

   //-- clean up simEventList --//
   simEventList.GoToHead();

   for ( node=0; node < simEventList.Length(); node++, ++simEventList )
      delete simEventList.Examine();

   //-- JOE what about the sysSimObj SIMOBJ struct???
   }



//-- SimEnv::RegisterSimObj() ----------------------------------
//
// Register an object on the simulation list
//
// gErrorCode = RE_REGISTER (register the object as an event) or
//              RE_NOREGISTER (don't register an event)
//-----------------------------------------------------------------

BOOL SimEnv::RegisterSimObj( SimObj* pSimObj, int registerFlag )
   {
   //--- first, build SimObjStruct ---//
   SIMOBJ *pSimObjStruct = new SIMOBJ;  // allocate space

   if ( pSimObjStruct == NULL )
      return RE_NOALLOC;

   pSimObjStruct->pSimObj = pSimObj;           // address of sim object

   double timeStep = pSimObj->GetTimeStep();  // time step
   long tickIncr = (long) ( (timeStep + 0.5*sysTimeStep) / sysTimeStep );

   if ( tickIncr < 1 )
      {
      //cout << "\nWarning! timeStep < sysTimeStep (setting to sysTimeStep)...\n
      pSimObj->SetTimeStep( sysTimeStep );
      tickIncr = 1;
      }

   int priority = pSimObj->GetPriority();             // priority

   pSimObjStruct->pNotifyFilter  = NULL;              // Notify objects Array

   simObjList.Append( pSimObjStruct );                // add to simObjList

   if ( registerFlag == RE_REGISTER )
      RegisterEvent( pSimObj, 0.0, priority, ET_DEFAULT, NULL );

   return TRUE;
   }

//-- SimEnv::RegisterSystemSimObj() ----------------------------
//
//  Sets up the SIMOBJ structure for this system simulation object
//-----------------------------------------------------------------
```

```
SIMOBJ *SimEnv::RegisterSystemSimObj( void )
   {
   //--- first, build SimObjStruct ---//
   SIMOBJ *pSimObjStruct = new SIMOBJ;  // allocate space

   pSimObjStruct->pSimObj = &sysSimObj;           // address of sim object

   double timeStep = sysSimObj.GetTimeStep();  // time step
   long tickIncr = (long) ( (timeStep + 0.5*sysTimeStep) / sysTimeStep );

   if ( tickIncr < 1 )
      {
      //cout << "\nWarning! timeStep < sysTimeStep (setting to sysTimeStep)..
      sysSimObj.SetTimeStep( sysTimeStep );
      tickIncr = 1;
      }

   //pSimObjStruct->tickIncr = tickIncr;

   //int priority = sysSimObj.GetPriority();             // priority
   //pSimObjStruct->priority = priority;

   pSimObjStruct->pNotifyFilter  = NULL;              // Notify objects Array

   return pSimObjStruct;
   }



//-- SimEnv::DeleteSimObj() ------------------------------------
//
//-- delete an object from the simulation list
//   (deletes any occurance from event list as well)
//-----------------------------------------------------------------

BOOL SimEnv::DeleteSimObj( SimObj* pSimObj )
   {
   SIMOBJ   *pSimObjStruct;
   int node;

   //--- scan simObjList for simulation object ---//
   simObjList.GoToHead();
   int simObjLength = simObjList.Length();

   for ( node=0; node < simObjLength; node++, ++simObjList )
      {
      pSimObjStruct = simObjList.Examine();

      if ( pSimObjStruct->pSimObj == pSimObj )  // item found??
         {
         delete pSimObjStruct;             // del SimObj structure
         simObjList.Delete();              // delete list node

         //-- next, delete all occurances of the object in the event list --/
         simEventList.GoToHead();
         int simEventLength = simEventList.Length();

         for ( node=0; node < simEventLength; node++ )
            {
```

```
            SIMEVENT *pSimEventStruct = simEventList.Examine();

            if ( pSimEventStruct->pSimObjStruct == pSimObjStruct )
               {                                  // event involves object??
               delete pSimEventStruct;            // delete event structure
               simEventList.Delete();             // delete event list node
               }

            ++simEventList;
            )  // end of event list

         ) // end of simObj found if

      else      //-- SimObj NOT found, so check notifyFilterArray --//
         {
         //-- get this simObj's NotifyFilterArray --//
         NotifyFilterArray *pNotifyFilterArray = \
                                 pSimObjStruct->pNotifyFilter;

         //- scan the array, looking at each NOTIFYFILTER's SimObjArray --//
         if ( pNotifyFilterArray != NULL )
            {
            for ( UINT i = 0; i < pNotifyFilterArray->Length(); i++ )
               {
               SimObjArray *pSimObjFilter = \
                           pNotifyFilterArray->Get(i)->pSimObjFilter;

               //-- does the simObj filter array exist? --//
               if ( pSimObjFilter != NULL )
                  {
                  UINT position = pSimObjFilter->Find( pSimObj );

                  if ( position != TA_NOTFOUND )         // if SimObj found,
                     pSimObjFilter->Delete( position );  // delete it

                  //-- anything left in the SimObjFilter? --//
                  if ( pSimObjFilter->Length() == 0 )
                     {
                     delete pSimObjFilter;
                     pNotifyFilterArray->Get(i)->pSimObjFilter = NULL;
                     }
                  )  // end of IF
               )  // end of FOR
            )  // end of IF (pNotifyFilter != NULL)
         )  // end of ELSE
      )  // end of main FOR loop

   return ErrorHandler( SC_SIMOBJ_NOT_FOUND );
   )


//-- SimEnv::Run() ------------------------------------------
//
// run clock from now until sysStopTime
//-----------------------------------------------------------------

int SimEnv::Run( void )
   {
   StartRun( );                  // start of simulation run
```

```
   SIMEVENT *pSimEventStruct;   // pointer to SIMEVENT structure
   SIMOBJ   *pSimObjStruct;     // pointer to SIMOBJ structure
   SimObj   *pSimObj;           // pointer to a simulation object
   int eventStatus;             // return value from Update() message
   int priority;
   long stopTick = (sysStopTime-sysStartTime+0.5*sysTimeStep) / sysTimeStep;

   //-- initialize timer variables --//
   sysCurrentTime = sysStartTime;
   sysCurrentTick = 0;

   //--- start simulation loop ---//
   while ( 1 )
      {
      YieldControl();   // defined for DOS or Windows

      //-- no more events on the event list? --//
      if ( simEventList.IsEmpty() )
         return EndRun( SC_NOEVENTS );

      simEventList.GoToHead();

      //-- get simulation event structure from top of event list --//
      pSimEventStruct = simEventList.Examine();

      //-- check for clock update --//
      if ( pSimEventStruct->eventTick > sysCurrentTick )
         {
         //-- calculate number of ticks to move forward --//
         long tickIncr = pSimEventStruct->eventTick - sysCurrentTick;

         sysCurrentTick = pSimEventStruct->eventTick;
         sysCurrentTime = sysStartTime + sysCurrentTick*sysTimeStep;

         sysCumTime     = sysCumTime + tickIncr*sysTimeStep;
         )

      //-- check for end of simulation --//
      if ( pSimEventStruct->eventTick >= stopTick )
         break;                        // return SC_EOS;

      //--- get simulation object address ---//
      pSimObjStruct = pSimEventStruct->pSimObjStruct;

      if ( pSimObjStruct )
         pSimObj = pSimObjStruct->pSimObj;
      else
         pSimObj = NULL;

      //-- check if trace is on --//
      if ( pSimObj && pSimObj->pTraceArray && traceProc )
         {
         TraceArray *pTraceData = pSimObj->InspectProperty();
         traceProc( this, pSimObj, pSimEventStruct->eventType, pTraceData );
         )

      //-- send Update message to the simobj --//
      eventStatus = pSimObj->Update( this, pSimEventStruct->eventType,
            pSimEventStruct->pData );
```

```
      //-- clean up event struct in event list --///
      simEventList.GoToHead();
      simEventList.Delete();           // remove event from head of list

      switch ( eventStatus )
         {
         case ES_TERMINATE:
            delete pSimEventStruct;    // deallocate memory for event struct
            return EndRun( SC_TERMINATED );   // exit the simulation

         case ES_UPDATE:      // re-register the event on the event list
            {
            //-- update next tick schedule ( stochastic or constant ) --//
            double timeStep = pSimObj->GetTimeStep();

            pSimEventStruct->eventTick = sysCurrentTick +
                  (long) ( (timeStep + 0.5*sysTimeStep) / sysTimeStep );

            priority = pSimObj->GetPriority();   // get priority

            //--- event built, now re-register it ---//
            RegisterEventStruct( pSimEventStruct, priority );
            }
            break;

         case ES_NOUPDATE:
         case ES_ERROR:
         default:
            delete pSimEventStruct;  // deallocate memory for event struct
            break;

         }     // end of SWITCH

      }  // end of WHILE

   //-- end of simulation - reset remaining event ticks and return     --//
   //-- NOTE: Any events from this run will be reset for the next run. --//
   //-- Assumes next run's startTime will be last run's stopTime       --//

   return EndRun( SC_EOS );      // end of simulation (sysStopTime) reached
   }


UINT SimEnv::StartRun( void )
  {
  // do some before run logic
  return 1;
  }


UINT SimEnv::EndRun( UINT terminateFlag )
   {
   SetStatisticFlags();

   switch ( terminateFlag )
      {
      case SC_EOS:
         {
         int eventCount = simEventList.Length();
         simEventList.GoToHead();

         for ( int counter=0; counter < eventCount; counter++ )
            {
            SIMEVENT *pSimEventStruct = simEventList.Examine();
            long stopTick = \
                  (sysStopTime-sysStartTime+0.5*sysTimeStep) / sysTimeStep;
            pSimEventStruct->eventTick -= stopTick;
            ++simEventList;
            }
         }
         break;

      case SC_NOEVENTS:
      case SC_TERMINATED:
      default:
         break;

      }  // end of SWITCH


   //-- print out user stats if user has defined a method --//
   if ( localReportProc )
      localReportProc( this );

   return terminateFlag;
   }


void SimEnv::StopSimulation( void )
   {
   RegisterEvent( NULL, 0, 0, ET_ENDSIMULATION, NULL );
   }


//-- SimEnv::RegisterEvent() ----------------------------------
//
// Register a new event on the event list, given an object, time and priority
// Note:  eventTime is floating point time interval from current time
//-------------------------------------------------------------------

BOOL SimEnv::RegisterEvent( SimObj* pSimObj, double eventIncr,
      int priority, UINT eventType, void *pData )
   {
   SIMOBJ *pSimObjStruct = NULL;

   if ( pSimObj == NULL )     // system event
      {
      //-- this logic dosen't handle possible RE_NOALLOC from this method --/
      //pSimObj = &sysSimObj;

      //??? JOE What the heck are you doing here!!!

      pSimObjStruct = RegisterSystemSimObj();

      if( pData == NULL )    // this event pertains to all simObjects
         pData = &sysSimObj;
      }

   else
      {
```

```
      //--- find simObj Ptr in simObjList ---//
      pSimObjStruct = simObjList.Find( pSimObj );

      //-- SimObj not found? --//
      if ( pSimObjStruct == NULL )
         return ErrorHandler( RE_NOSIMOBJ );

      ) // end of ELSE

   //--- allocate new event structure ---//
   SIMEVENT* pSimEventStruct = new SIMEVENT;

   if ( pSimEventStruct == NULL )
      return ErrorHandler( RE_NOALLOC ); // prob if SimObj was NULL

   //--- build event structure ---//
   pSimEventStruct->pSimObjStruct = pSimObjStruct;

   long eventTick = sysCurrentTick +  \
               (long) ( (eventIncr + 0.5*sysTimeStep) / sysTimeStep );

   pSimEventStruct->eventTick = eventTick;
   pSimEventStruct->eventType = eventType;
   pSimEventStruct->pData     = pData;

   RegisterEventStruct( pSimEventStruct, priority );

   return TRUE;
   }


//-- SimEnv::RegisterEventStruct() ------------------------------
//
//   Register an existing event struct on the event list, given the
//   structure's address and a priority
//-------------------------------------------------------------------

BOOL SimEnv::RegisterEventStruct( SIMEVENT *pSimEventStruct, int priority )
   {
   int priorityFlag = EP_PRIORITY_FOUND;

   simEventList.GoToHead();       // move to start of event list

   long nextTick = pSimEventStruct->eventTick;
   int tickFlag = SortEventTick( nextTick ); // sort by event timing

   //-- insert in Event List --//

   switch ( tickFlag )
      {
      case ET_TICK_FOUND_EQ:          // necessary to sort by priority
         priorityFlag = SortEventPriority( nextTick, priority );
         switch ( priorityFlag )
            {
            case EP_PRIORITY_FOUND:   // priority found, insert before
               simEventList.Insert( pSimEventStruct );
               break;

            case EP_EOL:     // end of list encountered, append
               simEventList.Append( pSimEventStruct );
               break;
            }
         break;

      case ET_TICK_FOUND_GT:
         simEventList.Insert( pSimEventStruct );
         break;

      case ET_EOL:                    // end of list encountered
         simEventList.Append( pSimEventStruct );
         break;
      }

   return TRUE;
   }


//-- SimEnv::SortEventTick() ------------------------------------
//
//   SortEventTick increments the event list up to the point where
//   an event is found which occurs at the same or later time
//
//   If: ET_TICK_FOUND_GT, the new event should be inserted BEFORE current
//       ET_TICK_FOUND_EQ, first equal tick is current, sort on priority
//       ET_EOL, event should be placed at end of the list
//-------------------------------------------------------------------

int SimEnv::SortEventTick( long nextTick )
   {
   SIMEVENT *pSimEventCurrent;    // current item in event list

   int length = simEventList.Length();
   simEventList.GoToHead();                // start at head of event list

   //--- get current event from eventList ---//
   for ( int nodes=0; nodes < length; nodes++ )
      {
      //--- get event pointer ---//
      pSimEventCurrent =  simEventList.Examine();

      //--- check timing ---//
      if ( pSimEventCurrent->eventTick > nextTick )
         return ET_TICK_FOUND_GT;

      else if ( pSimEventCurrent->eventTick == nextTick )
         return ET_TICK_FOUND_EQ;

      else
         ++simEventList;
      }

      return ET_EOL;   // end of list encountered
   }


//-- SimEnv::SortEventPriority() --------------------------------
//
//   SortEventPriority increments the event list up to the point where
//   an event is found has a lower priority than the update event
//
```

```
//   If: EP_PRIORITY_FOUND: event should be inserted BEFORE current item
//       EP_EOL:
//-------------------------------------------------------------------

int SimEnv::SortEventPriority( long eventTick, int priority )
   {
   SIMEVENT *pSimEventCurrent;    // current item in event list

   //--- get current event from eventList ---//

   //--- cycle though list ---//
   while( ! simEventList.IsTail() )
      {
      //--- get current event pointer from simEventList ---//
      pSimEventCurrent = simEventList.Examine();

      //--- does current event occur after new event? OR ---//
      if ( pSimEventCurrent->eventTick > eventTick )
         return EP_PRIORITY_FOUND;

      //--- same event time, new has higher priority? ---//
      else if ( pSimEventCurrent->pSimObjStruct->pSimObj->GetPriority()
                < priority )
         return EP_PRIORITY_FOUND;

      else
         ++simEventList;
      }

   //--- stopping point not found, so process tail ---//
   pSimEventCurrent = simEventList.Examine();

   if ( ( pSimEventCurrent->eventTick > eventTick )  \
     || ( pSimEventCurrent->pSimObjStruct->pSimObj->GetPriority() < priority ) )
      return EP_PRIORITY_FOUND;

   else
      return EP_EOL;
   }


//-- SimEnv::AddNotifyFilter() ---------------------------------
//
//-- adds a NOTIFYFILTER structure to a SimObj's NotifyFilterArray
//
//-- NOTE: the *pSimObjFilter is copied from the original.  The caller
//         is responsible for freeing any memory allocated for the
//         SimObjArray that is passed in.
//
//   AddNotifyFilter( pObjectToNotify, notifyType, pBroadcastingObjects );
//-------------------------------------------------------------------

BOOL SimEnv::AddNotifyFilter( SimObj *pObjectToNotify, UINT notifyType,
      SimObjArray *pBroadcastingObjects )
   {
   //-- scan SIMOBJ list for the specified SimObj --//
   SIMOBJ *pSimObjStruct = simObjList.Find( pObjectToNotify );

   if ( pSimObjStruct )   // item found?
      {
      NOTIFYFILTER *pNotifyFilter = new NOTIFYFILTER;

      pNotifyFilter->notifyType   = notifyType;

      //-- make copy of passed in SimObjArray filter --//
      SimObjArray *_pBroadcastingObjects = new SimObjArray( *pBroadcastingObj
      pNotifyFilter->pSimObjFilter = _pBroadcastingObjects;

      //-- does Notify Filter Array exist? --//
      if ( ! pSimObjStruct->pNotifyFilter )
         pSimObjStruct->pNotifyFilter = new NotifyFilterArray;

      //-- append NOTIFYFILTER struct to filter array --//
      pSimObjStruct->pNotifyFilter->Append( pNotifyFilter );

      return TRUE;         // exit out of simObjList scan
      }

   return FALSE;   // simObj not found
   }


//-- SimEnv::Broadcast() ---------------------------------------
//
//-- Notifies any interested SimObjs that a notification of type
//   notifyType has been sent by the pSender SimObject. Interested
//   SimObjects are objects that previously registerd interest in this
//   event when coming from the pSender SimObject. To register interest
//   in an event from pSender, a SimObject must send a message to the
//   SimEnv to AddNotifyFilter to itself.
//   Each SimObject can have an array of NotifyFilters. Each element in
//   the array has one event and one or more SimObjects.
//-------------------------------------------------------------------

BOOL SimEnv::Broadcast( SimObj *pSender, UINT notifyType, void **pData )
   {
   NOTIFY notifyStruct;

   notifyStruct.pSender    = pSender;
   notifyStruct.notifyType = notifyType;
   notifyStruct.pData      = pData;

   //--- get SIMOBJ struct from simObjList ---//
   simObjList.GoToHead();                  // start at head of event list

   for ( int i=0; i < simObjList.Length(); i++, ++simObjList )
      {
      //--- get SIMOBJ pointer ---//
      SIMOBJ *pSimObjStruct = simObjList.Examine();

      NotifyFilterArray *pNotifyFilter = pSimObjStruct->pNotifyFilter;

      //-- now have pointer to NotifyFilterArray.  if NULL, this object
      //-- doesn't want to be notified about any events, so continue.
      //-- otherwise, look through the NotifyFilterArray for NOTIFYFILTERS
      //-- that contain this eventType...

      if ( pNotifyFilter )
         {
         UINT index = pNotifyFilter->FindFilter( notifyType );
```

```
         if ( index != TA_NOTFOUND )  // did we find something??
            {
            //-- this SimObj is interested in this event.  Next step is to
            //-- determine if it is interested in the object which will
            //-- handle the event.

            SimObjArray *pSimObjArray = \
                        pNotifyFilter->Get( index )->pSimObjFilter;

            if ( ( pSimObjArray == NULL ) // interested regardless of source
              || ( pSimObjArray->Find( notifyStruct.pSender )// pReceiverSimObj
                   != TA_NOTFOUND ) )
               {
               //-- notify object --//
               BOOL notifyHandled = \
                     pSimObjStruct->pSimObj->Notify( this, &notifyStruct );

               if ( notifyHandled )
                  {
                  pData = notifyStruct.pData;
                  return TRUE;
                  }
               }
            }
         }
      }

   return FALSE;
   }


void SimEnv::SetStatisticFlags( void )
   {
   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for ( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      distStats       += pSimObj->IsDistsOn();
      obsStats        += pSimObj->IsObsStatsOn();
      continStats     += pSimObj->IsContinStatsOn();
      timeStats       += pSimObj->IsTimeStatsOn();
      countStats      += pSimObj->IsCountStatsOn();
      }

   return;
   }


void SimEnv::StartTracingAllObjects( void )
   {
   simObjList.GoToHead();
   int simObjListLength = simObjList.Length();

   for( UINT i = 0; i < simObjListLength; i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      pSimObjStruct->pSimObj->StartTrace();
      }

   return;
   }


void SimEnv::StopTracingAllObjects( void )
   {
   simObjList.GoToHead();
   int simObjListLength = simObjList.Length();

   for( UINT i = 0; i < simObjListLength; i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      pSimObjStruct->pSimObj->StopTrace();
      }

   return;
   }


void SimEnv::EnumEvents( EVENTPROC eventFn )
   {
   if ( eventFn == NULL )
      return;

   //-- goto head of SimEvent list --//
   simEventList.GoToHead();
   UINT length = simEventList.Length();

   //-- send message indicating start of listing --//
   eventFn( NULL, sysCurrentTime, 0, 0 );

   for( UINT i=0; i < length; i++, ++simEventList )
      {
      SIMEVENT *pSimEventStruct = simEventList.Examine();
      SIMOBJ   *pSimObjStruct   = pSimEventStruct->pSimObjStruct;
      SimObj   *pSimObj         = pSimObjStruct->pSimObj;
      double    eventTime       = \
                     ((double) pSimEventStruct->eventTick * sysTimeStep );
      UINT      priority        = pSimObj->GetPriority();
      UINT      eventType       = pSimEventStruct->eventType;

      //-- make call to enum function --//
      eventFn( pSimObj, eventTime, priority, eventType );

      ) // end of FOR

   //-- send message indicating end of listing --//
   eventFn( NULL, sysCurrentTime, 0, ET_ENUMEVENTLIST );

   return;
   }



//-- during replications we must reset some system state --//
void SimEnv::ClearSystemState()
```

```
    {
    //-- set the clock back to its almost initial state --//

    ClearSimObjList();
    sysCurrentTime = sysStartTime;
    sysCumTime = sysStartTime;
    sysCurrentTick = 0;
    ClearEventList();
    ClearStatisticFlags();
    }


void SimEnv::GetStatisticLengths( UINT *pDistLength, UINT *pObsLength,
     UINT *pTimeLength, UINT *pContinLength, UINT *pCountLength )
    {
    SIMOBJ              *pSimObjStruct;

    DistArray           *pDistArray;
    StatsObsArray       *pObsArray;
    StatsTimeArray      *pTimeArray;
    StatsContinArray    *pContinArray;
    CounterArray        *pCounterArray;

    simObjList.GoToHead();
    UINT simObjLength = simObjList.Length();

    for ( UINT i = 0; i < simObjLength; i++, ++simObjList )
       {
       pSimObjStruct = simObjList.Examine();


       //-- sum DistArray length --//

       pDistArray = (DistArray*)
            pSimObjStruct->pSimObj->IsDistsOn();

       if( pDistArray )
          *pDistLength = *pDistLength + pDistArray->Length();

       //-- sum ObsArray length --//
       pObsArray = (StatsObsArray*) pSimObjStruct->pSimObj->IsObsStatsOn();

       if( pObsArray )
          *pObsLength = *pObsLength + pObsArray->Length();


       //-- sum TimeArray length --//

       pTimeArray = (StatsTimeArray*) pSimObjStruct->pSimObj->IsTimeStatsOn();

       if( pTimeArray )
          *pTimeLength = *pTimeLength + pTimeArray->Length();


       //-- sum ContinuousArray length --//

       pContinArray = (StatsContinArray*)
            pSimObjStruct->pSimObj->IsContinStatsOn();

       if( pContinArray )
          *pContinLength = *pContinLength + pContinArray->Length();


       //-- sum CounterArray length --//

       pCounterArray = (CounterArray*)
            pSimObjStruct->pSimObj->IsCountStatsOn();

       if( pCounterArray )
          *pCountLength = *pCountLength + pCounterArray->Length();

       }

    return;
    }


void SimEnv::ClearStatisticFlags( void )
    {
    distStats       = FALSE;
    obsStats        = FALSE;
    timeStats       = FALSE;
    continStats     = FALSE;
    countStats      = FALSE;

    return;
    }


void SimEnv::ReInitializeAllStats( double _currentTime )
    {
    lastClearStatsTime = _currentTime;

    SIMOBJ *pSimObjStruct;

    //-- go through the simObjList and send a message to all the objects.

    simObjList.GoToHead();
    UINT simObjLength = simObjList.Length();

    for ( UINT i = 0; i < simObjLength; i++, ++simObjList )
       {
       pSimObjStruct = simObjList.Examine();

       pSimObjStruct->pSimObj->ReInitializeAllStats( _currentTime );
       }

    return;
    }


UINT NotifyFilterArray::FindFilter( UINT notifyType )
    {
    if ( length == 0 )
       return TA_NOTFOUND;

    for ( UINT i=0; i < length; i++ )
       {
```

```
      if ( array[ i ]->notifyType == notifyType )
         return i;
      }

   return TA_NOTFOUND;
   }


SIMOBJ *SIMOBJList::Find( SimObj *pSimObj )
   {
   GoToHead();

   for ( int node=0; node < length; node++ )
      {
      SIMOBJ *pSimObjStruct = Examine();

      if ( pSimObjStruct->pSimObj == pSimObj )
         return pSimObjStruct;
      else
         GoToNext();
      }

   return NULL;
   }


void YieldControl( void )
   {

#if defined ( _Windows )
   //-- see if any windows messages are waiting in the app queue --//
   MSG  msg;
   while( PeekMessage( &msg, NULL, NULL, NULL, PM_REMOVE ) )
      {
      //-- check for IDM_STOPs (and ViewBox??) messages --//
      //-- dump all others (should add more later!!) --//
      //if ( msg.message == IDM_STOP )
         {
         TranslateMessage( &msg );
         DispatchMessage( & msg );
         }
      }
#endif

   return;
   }



//---------------------- Integration Methods ---------------------//


//-- IntRKS() ------------------------------------------------------
//
//--- general purpose Runge-Kutta integrator (fourth order)
//    code is designed and modified from "Applied Numerical Methods" by
//    Brice Carnahan, H.A. Luther, and James O. Wilkes., 1969., pgs 374-375.
//
//--INPUTS
//   int    svCount    = state variable count
//   float  stateVar   = state variable value
//   double derivative = derivative value
//   float  *time      = time variable address
//   float  dt         = timestep
//-----------------------------------------------------------------------

/*
---------------------------------------------------------------
Example SimObj::State() method

void MySimObj::State( int svCount, double* stateVar, double* derivative, doub
   {
   sv02 = stateVar[ 0  ];
   etc....

   d02_dt = ... etc.
   .. etc...

   // derivative[ 0 ] = d02_dt;
   // ...etc...
   }
---------------------------------------------------------------
*/

BOOL IntRK$(
      SimObj *pSimObj,
      double timeStep,
      double currentTime,
      int    svCount,
      double *stateVar,         // state variable array
      STATEPROC lpFn
      )
   {
   const int totalPasses = 4;
   int  i;                      // loop counter

   double *derivative = new double[ svCount ];
   double *svTemp     = new double[ svCount ];
   double *phi        = new double[ svCount ];      // increment function

   for ( int pass = 0; pass < totalPasses; pass++ )
      {
      //-- get derivative and state values from object --//
      if ( lpFn )
         (pSimObj->*lpFn)( svCount, stateVar, derivative, currentTime );
      else
         pSimObj->State( svCount, stateVar, derivative, currentTime );

      //-- select appropriate RKS stage --//
      switch ( pass )
         {
         case 0:
            for ( i=0; i < svCount; i++ )
               {
               svTemp[ i ]   = stateVar[ i ];
               phi[ i ]      = derivative[ i ];
               stateVar[ i ] = svTemp[ i ] + \
                                    ( 0.5 * timeStep * derivative[ i ] );
               }
```

```
            //-- increment temporary time variable --//
            currentTime += ( 0.5 * timeStep );
            break;

         case 1:
            for ( i=0; i < svCount; i++ )
               {
               phi[i]       += ( 2.0 * derivative[ i ] );
               stateVar[ i ] = svTemp[ i ] + \
                               ( 0.5 * timeStep * derivative[ i ] );
               }
            break;

         case 2:
            for ( i=0; i < svCount; i++ )
               {
               phi[i]     += ( 2.0 * derivative[ i ] );
               stateVar[i] = svTemp[ i ] + ( timeStep * derivative[ i ] );
               }

            //-- increment temporary time variable --//
            currentTime += ( 0.5 * timeStep );
            break;

         case 3:
            for ( i=0; i < svCount; i++ )
               {
               stateVar[i] = svTemp[i] + \
                        ( (phi[ i ] + derivative[ i ]) * timeStep / 6.0 );
               }
            break;

         } // end of pass switch

      } // end of for loop

   //-- clean up --//
   delete derivative;
   delete svTemp;
   delete phi;

   return TRUE;
   }


/*
-----------------------------------------------------------------
THE NEXT INTEGRATOR TO IMPLEMENT!

Runge-Kutta-Fehlberg Formula  (RKF45) 1969
from "An Introduction to Numerical Computations by Sidney Yakowitz and
Ferenc Szidarovszky 1986. pg. 324.

Based on the six seperate calls to subroutine State, (RKF45) computes
an estimate for each state variable for one timeStep, an estimate of
the error for this step, and a new proposed size for the next step.

The step is usually specified with minimum and maximum values. (RKF45)
tries to use the largest stepsize, but the next proposed size may be
smaller in order to meet the minimum desired accuracy for the state
variables.

If the error is too large, the integration is repeated with a smaller
stepsize.
-----------------------------------------------------------------
*/



/*
-----------------------------------------------------------------
int main()
   {
   SimEnv SimEnv;
   ChildSimObj simObj1("Object 1", 0.1, 0);
   ChildSimObj simObj2("Object 2", 0.2, 1);

   SimEnv.RegisterSimObj( &simObj1, RE_REGISTER );
   SimEnv.RegisterSimObj( &simObj2, RE_REGISTER );

   SimEnv.Run();
   return 0;
   }
-----------------------------------------------------------------
*/
```

```
Print date: 8/17/1992     Print time:  8:40
//-------------------------------------------------------------------
//  PROGRAM: STSIMENV.HPP
//
//  PURPOSE: Class declaration for the Stat Simulation Environment object
//
//  COLLABORATORS: SimObj, TDList
//-------------------------------------------------------------------

#if !defined  _STSIMENV_HPP
#define       _STSIMENV_HPP 1

//------------------------- Includes ---------------------------//

#if !defined ( _SIMENV_HPP )
#include <simenv.hpp>
#endif


#if !defined ( _SIMGLOBA_HPP )
#include "simgloba.hpp"
#endif

class SimEnvGlobalStats;

//------------------------- Constants --------------------------//


class StatSimEnv : public SimEnv
   {
   friend SimEnvGlobalStats;   // for now

   private:
      //-- global statistics object --//
      SimEnvGlobalStats *pGlobalStats;

      //-- run pre and post processing --//
      virtual UINT StartRun( void );
      virtual UINT EndRun( UINT terminateFlag );

   protected:
      //-- replication data --//
      UINT numReps;

      //-- moved from simgloba.hpp --//
      UINT repNum;

      //-- statistic flags --//
      virtual void ClearStatisticFlags( void );
      UINT globalStats;
      FLAG replicatingFlag;

      void (*globalReportProc)( StatSimEnv* );

      //void SetStatisticFlags( void );

   public:
      //-- constructors --//
      StatSimEnv( char *_name );
      StatSimEnv( char *_name, double startTime, double stopTime,\
            double timeStep, SIMOBJList simObjects );

      //-- destructor --//
      ~StatSimEnv( void );

      //-- replication info --//
      UINT GetNumReps( void )          { return numReps; }
      void Replicate( UINT _numReps, REPPROC pRepFn );
      void RecordGlobalStatistics( void );
      SimEnvGlobalStats *GetGlobalStatsPtr( void ) { return pGlobalStats; }

      //int Run( void );

      //-- enumerate observational statistics to a user defined function --//
      virtual BOOL GlobalStatsAvailable( void ) { return (BOOL) globalStats;

      void SetGlobalReportProc( void (*reportProc)(StatSimEnv*) )
            { globalReportProc = reportProc; }

      void EnumGlobalDistStats        ( REPORTPROC );
      void EnumGlobalObsStats         ( REPORTPROC );
      void EnumGlobalContinStats      ( REPORTPROC );
      void EnumGlobalTimeStats        ( REPORTPROC );
      void EnumGlobalCountStats       ( REPORTPROC );
   };


//---------------------------- Inlines --------------------------//

#endif
```

```
Print date: 8/17/1992     Print time:  8:44
//-------------------------------------------------------------------
//  PROGRAM: StSimEnv.cpp
//-------------------------------------------------------------------

#include <windows.h>  // for eventListWindow messaging only!!
#pragma hdrstop

//------------------------ Includes ----------------------------//

#include <tdlist.hpp>
#include <simobj.hpp>
#include <stsimenv.hpp>

#include <typedefs.hpp>

#include <limits.h>


#if !defined ( __STDIO_H )
#include <stdio.h>
#endif


//------------------------ Static data -------------------------//


//------------------------ Externals ---------------------------//


//------------------------ C Functions -------------------------//

/*
-------------------------------------------------------------------
   The StatSimEnv ( simulation clock ) is designed for discrete or
   continuous simulations involving replications.
-------------------------------------------------------------------
*/



//-- StatSimEnv::StatSimEnv() -------------------------------------
//
//-- constructor methods
//-----------------------------------------------------------------

//--- No arguments (simObjList empty) ---//
StatSimEnv::StatSimEnv( char *_name )
   : SimEnv          ( _name ),
     pGlobalStats    ( NULL ),
     numReps         ( 0 ),
     repNum          ( 0 ),
     globalStats     ( FALSE ),
     replicatingFlag ( FALSE ),
     globalReportProc( NULL )
   {
   }


StatSimEnv::StatSimEnv( char *_name, double startTime, double stopTime,\
                        double timeStep, SIMOBJList simObjects )
   : SimEnv( _name, startTime, stopTime, timeStep, simObjects ),
     pGlobalStats     ( NULL ),
     numReps          ( 0 ),
     repNum           ( 0 ),
     globalStats      ( FALSE ),
     replicatingFlag  ( FALSE ),
     globalReportProc ( NULL )
   {
   }


//-- StatSimEnv::~StatSimEnv() ------------------------------------
//
//-----------------------------------------------------------------

StatSimEnv::~StatSimEnv( void )
   {
   //-- clean up global statistics --//
   if ( pGlobalStats )
      delete pGlobalStats;
   }


UINT StatSimEnv::StartRun( void )
  {
  // do some before run logic
  SimEnv::StartRun();       // run in parent

  //-- reseed sim objects --//
//  if ( pGlobalStats->repNum > 0 )
  if ( repNum > 0 )   // if this is not the first rep !!
     {
     DistArray    *pDistArray;

     simObjList.GoToHead();
     UINT simObjLength = simObjList.Length();

     for ( UINT i = 0; i < simObjLength; i++, ++simObjList )
        {
        SIMOBJ *pSimObjStruct = simObjList.Examine();

        //-- set Dist seeds --//
        pDistArray = (DistArray*) pSimObjStruct->pSimObj->IsDistsOn();

        if ( pDistArray )
           pGlobalStats->SetGlobalDistSeeds( pDistArray );
        }

     } // end of IF

  else   // first replication
     {
     }   // end of ELSE

  return 1;
  }


UINT StatSimEnv::EndRun( UINT terminateFlag )
```

```
    {
    SimEnv::EndRun( terminateFlag );    // run in parent

    if ( replicatingFlag )
       {
       //-- new Globals statistics object and record stats.... --//
       RecordGlobalStatistics();  // also stores last seeds
       }


    //-- print out global stats if user has defined a method and
    //   this is the last run.

//    if ( globalReportProc && (numReps - 1) == pGlobalStats->repNum )
    if ( globalReportProc && (numReps - 1) == repNum )
       {
       //-- set flags for output of statistics --//
       globalStats     = TRUE;
       distStats       = FALSE;  // we don't need the single run stats anymore
       obsStats        = FALSE;
       timeStats       = FALSE;
       continStats     = FALSE;
       countStats      = FALSE;

       pGlobalStats->SetUpGlobalStatsOutput( numReps );

       globalReportProc( this );
       }

    return terminateFlag;
    }



void StatSimEnv::Replicate( UINT _numReps, REPPROC pRepFn )
   {
   numReps = _numReps;

   replicatingFlag = TRUE;

   for ( UINT i = 0; i < numReps; i++ )
      {
      //-- initialize each rep --//
//      pGlobalStats->repNum = i;
      repNum = i;

      //-- run an experiment --//
      pRepFn( this );

      //-- here's a hack!

      if ( i != numReps - 1 )
         ClearSystemState();
//      else
//         MyHeaderOutput( this );
      }
/*
--------
   //-- set flags for output of statistics --//
   globalStats     = TRUE;
   distStats       = FALSE;  // we don't need the single run stats anymore
   obsStats        = FALSE;
   timeStats       = FALSE;
   continStats     = FALSE;
   countStats      = FALSE;

   pGlobalStats->SetUpGlobalStatsOutput( numReps );
--------
*/

   )


// also stores last seeds
void StatSimEnv::RecordGlobalStatistics( void )
   {
   if ( ! pGlobalStats )   // 1st time recording stats
      {
      pGlobalStats = new SimEnvGlobalStats( this );


      UINT distLength       = 0;
      UINT obsLength        = 0;
      UINT timeLength       = 0;
      UINT continLength     = 0;
      UINT countLength      = 0;

      //-- go through the SimObj list and get a count of the space that
      //   will be needed to store all the information.

      GetStatisticLengths( &distLength, &obsLength, &timeLength,
            &continLength, &countLength );

      //-- set up arrays to record global statistics --//
      pGlobalStats->Setup( numReps, distLength, obsLength,
            timeLength, continLength, countLength );


    DistArray    *pDistArray;

    //-- record the initial seeds in the global statistics --//
    simObjList.GoToHead();
    UINT simObjLength = simObjList.Length();

    for ( UINT i = 0; i < simObjLength; i++, ++simObjList )
       {
       SIMOBJ *pSimObjStruct = simObjList.Examine();

       //-- set Dist seeds --//
       pDistArray = (DistArray*) pSimObjStruct->pSimObj->IsDistsOn();

       if ( pDistArray )
          pGlobalStats->SetInitialSeeds( pDistArray );
       }

     )  // end of IF


 SIMOBJ            .    *pSimObjStruct;
```

```
    DistArray           *pDistArray;
    StatsContinArray    *pContinArray;
    StatsTimeArray      *pTimeArray;
    StatsObsArray       *pObsArray;
    CounterArray        *pCounterArray;

    simObjList.GoToHead();
    UINT simObjLength = simObjList.Length();

    for ( UINT i = 0; i < simObjLength; i++, ++simObjList )
       {
       pSimObjStruct = simObjList.Examine();


       //-- set Dist statistics --//

       pDistArray = (DistArray*)
             pSimObjStruct->pSimObj->IsDistsOn();

       if ( pDistArray )
          pGlobalStats->SetGlobalDistStats( pDistArray );


       //-- set Continuous statistics --//

       pContinArray = (StatsContinArray*)
             pSimObjStruct->pSimObj->IsContinStatsOn();

       if ( pContinArray )
          pGlobalStats->SetGlobalContinStats( pContinArray );


       //-- set Time statistics --//

       pTimeArray = (StatsTimeArray*)
             pSimObjStruct->pSimObj->IsTimeStatsOn();

       if ( pTimeArray )
          pGlobalStats->SetGlobalTimeStats( pTimeArray );


       //-- set Observational statistics --//

       pObsArray = (StatsObsArray*)
             pSimObjStruct->pSimObj->IsObsStatsOn();

       if ( pObsArray )
          pGlobalStats->SetGlobalObsStats( pObsArray );


       //-- set Count statistics --//

       pCounterArray = (CounterArray*) pSimObjStruct->pSimObj->IsCountStatsOn();

       if ( pCounterArray )
          pGlobalStats->SetGlobalCountStats( pCounterArray );

       }  // end of FOR

    //-- set SimEnvGlobalStats array row counters back to zero --//

       pGlobalStats->distRow   = 0;
       pGlobalStats->continRow = 0;
       pGlobalStats->timeRow   = 0;
       pGlobalStats->obsRow    = 0;
       pGlobalStats->countRow  = 0;
       pGlobalStats->seedRow   = 0;

       return;
       }

    void StatSimEnv::ClearStatisticFlags( void )
       {
       SimEnv::ClearStatisticFlags();

       globalStats = FALSE;

       return;
       }
```

Appendix H: Random number generator classes

```
Print date: 8/17/1992     Print time:  8:38
//---------------------------------------------------------------------
// PROGRAM: generate.cpp
//
// By: Joseph A Fisher
//     Oregon State University
//     fisherj@ucs.orst.edu
//
// Date: April 1992
// Main program to generate random numbers to an output file.
//---------------------------------------------------------------------

#pragma hdrstop

//--------------------------- Includes --------------------------//
#include <stdio.h>
#include <iostream.h>

#include <random.hpp>


int main( int argc, char* argv[ ] )
   {
   //-- open the random.dat output data file for writing --//
   FILE *pOutput1 = fopen( "random1.dat", "wt" );
   FILE *pOutput2 = fopen( "random2.dat", "wt" );
   if ( pOutput1 == NULL || pOutput2 == NULL )
      {
      cout << "Error opening random1.dat or ramdom2.dat output data file";
      return 0;
      }

   //-- create every possible type of random number object --//
   RandUniform          RNG1  ( 0.0, 1.0 );
   RandExponential      RNG2  ( 5.5 );
   RandTriangular       RNG3  ( 1.0, 5.0, 10.0 );
   RandWeibull          RNG4  ( 3.0, 6.0 );
   RandPoisson          RNG5  ( 6.0 );
   RandErlang           RNG6  ( 5, 5.0 );
   RandBernoulli        RNG7  ( 0.65 );
   RandNormal           RNG8  ( 5.0, 1.0, 0 );
   RandLogNormal        RNG9  ( 10.0, 3.5, 0 );
   RandBeta             RNG10 ( 2.1, 4.1, 0, 1 );
   Rand_F               RNG11 ( 4, 40, 1, 0 );
   RandDiscreteUniform RNG12 ( (long) 5, 10, 0 );
   RandBinomial         RNG13 ( 20, 0.1 );
   Rand_T               RNG14 ( 10, 0 );
   RandChiSquare        RNG15 ( 4, 0 );
   RandGamma            RNG16 ( 0.85, 0.4 );
//   RandGamma            RNG16 ( 0.6, 3.2 );
   RandGeometric        RNG17 ( 0.25 );
   RandNegBinomial      RNG18 ( 5, 0.5 );


   //-- print the headers --//
   fprintf( pOutput1, "%-25s" "%-25s" "%-25s" "%-25s" "%-25s" "%-25s"
                      "%-25s" "%-25s" "%-25s\n", RNG1.GetHeader(),
                      RNG2.GetHeader(), RNG3.GetHeader(), RNG4.GetHeader(),
                      RNG5.GetHeader(), RNG6.GetHeader(), RNG7.GetHeader(),
                      RNG8.GetHeader(), RNG9.GetHeader() );

   //-- print the headers --//
   fprintf( pOutput2, "%-25s" "%-25s" "%-25s" "%-25s" "%-25s" "%-25s"
                      "%-25s" "%-25s" "%-25s\n", RNG10.GetHeader(),
                      RNG11.GetHeader(), RNG12.GetHeader(), RNG13.GetHeader(),
                      RNG14.GetHeader(), RNG15.GetHeader(), RNG16.GetHeader(),
                      RNG17.GetHeader(), RNG18.GetHeader() );


   //-- generate random numbers --//
   for( UINT i = 0; i < 1000; i++ )
      {
      //-- print out the random numbers --//
      fprintf( pOutput1, "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f"
                         "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f\n",
               RNG1.RandValue(), RNG2.RandValue(), RNG3.RandValue(),
               RNG4.RandValue(), RNG5.RandValue(), RNG6.RandValue(),
               RNG7.RandValue(), RNG8.RandValue(), RNG9.RandValue() );


      //-- print out the random numbers --//
      fprintf( pOutput2, "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f"
                         "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f" "%-25.10f\n",
               RNG10.RandValue(), RNG11.RandValue(), RNG12.RandValue(),
               RNG13.RandValue(), RNG14.RandValue(), RNG15.RandValue(),
               RNG16.RandValue(), RNG17.RandValue(), RNG18.RandValue() );
   }

//-- close the output file --//
fclose( pOutput1 );
fclose( pOutput2 );

return 1;
}
```

```
Print date: 8/17/1992     Print time:  8:26
//---------------------------------------------------------------------
// PROGRAM: rand.hpp
//
// PURPOSE: Generates uniform ( 0, 1 ) random numbers
//
// NOTE:    "Intuitively, the random variables X and Y ( whether
//          discrete or continuous ) are independent if knowing the
//          value that one random variable takes on tells us nothing
//          about the distribution of the other. Also, if X and Y are
//          not independent, we say they are dependent."
//          [ Law & Kelton, 1991, pg. 275 ]
//---------------------------------------------------------------------

/*
---------------------------------------------------------------------
This generator should have the following properties:
   1. The computer code is fast and uses a small amount of memory
   2. It generates repeatable sequences
   3. The resulting values pass resonable tests for being from the
      uniform distribution
   4. The resulting sequences pass a number of different tests for randomness.
   5. A lattice structure is not obviously present.
   [ Thesen and Travis, 1992 ]
   6. Portability

   This generator implements the recommendations of Peter W. Lewis for
prime modulus multiplicative congruential random number generators (PMMCG).
[ Lewis and Orav, 1989 ]

   The formula is: 2^31 - 1 = LONG_MAX = 2,147,483,647
                   seed(i+1) = ( multiplier*seed(i) ) mod( LONG_MAX )
                   U(0,1)    = seed(i+1) / LONG_MAX;

   Note: The seed should never be 0 or LONG_MAX!

   FishMan & Moore find the 5 optimal full period multipliers for the
(PMMCG) with prime modulus 2^31 -1. They present a battery of statistical
tests for evidence. They also discuss some poor results of some previously
used multipliers. [ FishMan and Moore, 1986 ]

FishMan & Moore propose the following 5 as the best multipliers:
      multiplier  =   950,706,376
                  =   742,938,285
                  = 1,226,874,159
                  =    62,089,911
                  = 1,343,714,438

   If the user dosen't select a starting seed, I must provide a unique
default starting seed for them. Since I want many different default
starting seeds, I will use the static 'defaultSeed' to increment the
starting default seeds. The range of initial seeds is just over 2 billion,
so I'll space them apart by 1 million and get 2,000 different streams
for each multiplier. The length of each stream before overlapping is
1 million ( I don't know, is it? ). Using all 5 different multipliers will
give 5 * 2,000 = 10,000 different starting seeds.


STREAMS IMPLEMENTED
//---------------- 1 - 5 are recommend ! ------------//

 1  FishMan & Moore 1 (1985)   (MCG) a =   950,706,376
 1  IMSL3(1987)                (MCG) a =   950,706,376
 2  FishMan & Moore 2 (1985)   (MCG) a =   742,938,285
 3  FishMan & Moore 3 (1985)   (MCG) a = 1,226,874,159
 4  FishMan & Moore 4 (1985)   (MCG) a =    62,089,911
 5  FishMan & Moore 5 (1985)   (MCG) a = 1,343,714,438

 6  SIMSCRIPT II.5 (1983)      (MCG) a =   630,360,016
 7  GPSS/PC(1974)              (MCG) a =   397,204,094
 7  SAS(1982)                  (MCG) a =   397,204,094
 7  IMSL2(1984)                (MCG) a =   397,204,094
 8  SPSS(1983)                 (MCG) a =        16,807
 8  APL(1970)                  (MCG) a =        16,807
 8  IMSL1(1984)                (MCG) a =        16,807

References: Simulation for Decision Making by Arne Thesen and
            Laurel E. Travis, 1992, 372-373.

            Simulation methodology for Statisticians, Operations Analysts,
            and Engineers by Peter Lewis and E. Orav, Volume 1, 1989.,
            pg., 89.

            An exhaustive analysis of multiplicative congruential random
            number generators with modulus 2^31 - 1" by George S. Fishman
            and Louis R. Moore III, SIAM J. Sci. Stat. Comput, Vol. 7,
            No. 1, January 1986, pgs., 24 - 45.
---------------------------------------------------------------------
*/

#if !defined _RAND_HPP
#define      _RAND_HPP 1


//------------------------ Typedefs ----------------------------//

typedef unsigned int  UINT;
typedef unsigned long ULONG;

#define  FALSE 0
#define  TRUE  1

//------------------------ Constants ---------------------------//

const int CAPTIONLENGTH = 32;


class Rand
  {

  //-- Prime modulus multiplicative congruential generator --//
  private:

     static long defaultSeed;          // increments seeds by 1,000,000

     long seed;                        // current RNG seed
     long initialSeed;                 // initial RNG seed
     long multiplier;                  // RNG multiplier
     long observations;                // # of times this generator was used
     char *name;                       // statistic name
```

```
    char *distName;                 // generator name
    char *parameterA;               // parameter a stored as a string
    char *parameterB;               // parameter b " "
    char *parameterC;               // parameter c " "
    char *header;                   // eg. "Uniform(0.0,1.0)"

    void SetDefaultSeed( void );
    long GetDefaultSeed( void );
    void SetInitialSeed( long _seed );  // can only be set once!

    void SelectMultiplierType ( UINT _multiplier );
    void SelectMultiplierType1( void ) { multiplier =  950706376; }
    void SelectMultiplierType2( void ) { multiplier =  742938285; }
    void SelectMultiplierType3( void ) { multiplier = 1226874159; }
    void SelectMultiplierType4( void ) { multiplier =   62089911; }
    void SelectMultiplierType5( void ) { multiplier = 1343714438; }
    void SelectMultiplierType6( void ) { multiplier =  630360016; }
    void SelectMultiplierType7( void ) { multiplier =  397204094; }
    void SelectMultiplierType8( void ) { multiplier =      16807; }

protected:

    //-- constructors --//
    Rand( UINT _stream, char *_name = "-" );
    Rand( UINT _stream, long _seed, char *_name = "-" );

    //-- get U( 0,1 ) random number from generator --//
    double GetUnif01( void );     // also increments observations

    void AddOneObservation( void ) { observations++; }

public:

    //-- destructor --//
    virtual ~Rand( void );

    //-- pure virtual, child must return random number --//
    virtual double RandValue( void ) = 0;

    //-- reset the statistics of the RNG --//
    void ReInitializeStats( void ) { observations = 0; }

    //-- Get: output information --//
    const char* const GetName( void ) { return (const char* const) name; }
    const char* const GetDistName( void )
          { return (const char* const) distName; }
    const char* const GetA( void ) { return (const char* const) parameterA; }
    const char* const GetB( void ) { return (const char* const) parameterB; }
    const char* const GetC( void ) { return (const char* const) parameterC; }
    const char* const GetHeader( void );

    void SetName           ( char *_name );
    void SetDistName       ( char *_name );
    void SetParameterA_Name( char *_name );
    void SetParameterB_Name( char *_name );
    void SetParameterC_Name( char *_name );

    long GetInitialSeed( void )  { return initialSeed; }
    long GetMultiplier( void )   { return multiplier; }
    long GetObservations( void ) { return observations; }

    long GetSeed( void )         { return seed; }

    void SetSeed( long _seed );

};

#endif
```

```
Print date: 8/17/1992     Print time:  8:36
//---------------------------------------------------------------------
//  PROGRAM: rand.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

#if !defined ( __MATH_H )
extern "C"
{
#include <math.h>
}
#endif


extern "C"
{
#include <string.h>
}

const long LONG_MAX=2147483647;

long Rand::defaultSeed = 0;             // static to entire class
                                        // which means every instance

//-- Rand::Rand() ------------------------------------------------
//
// constructors
//---------------------------------------------------------------------

Rand::Rand( UINT _multiplierType, char *_name )
   : seed( 0 ),
     initialSeed( 0 ),
     multiplier( 0 ),
     observations( 0 ),
     name( NULL ),
     distName( NULL ),
     parameterA( NULL ),
     parameterB( NULL ),
     parameterC( NULL ),
     header( NULL )
   {
   SetName( _name );
   SetDistName( "-" );
   SetParameterA_Name( "-" );
   SetParameterB_Name( "-" );
   SetParameterC_Name( "-" );

   SelectMultiplierType( _multiplierType );
   SetDefaultSeed();
   }


Rand::Rand( UINT _multiplier, long _seed, char *_name )
   : seed( 0 ),
     initialSeed( 0 ),
     multiplier( 0 ),
     observations( 0 ),
     name( NULL ),
     distName( NULL ),
     parameterA( NULL ),
     parameterB( NULL ),
     parameterC( NULL ),
     header( NULL )
   {
   SetName( _name );
   SetDistName( "-" );
   SetParameterA_Name( "-" );
   SetParameterB_Name( "-" );
   SetParameterC_Name( "-" );

   SelectMultiplierType( _multiplier );
   SetSeed( _seed );
   }


Rand::~Rand( void )
   {
   if ( name )
      delete name;

   if ( distName )
      delete distName;

   if ( parameterA )
      delete parameterA;

   if ( parameterB )
      delete parameterB;

   if ( parameterC )
      delete parameterC;

   if ( header )
      delete header;
   }


const char* const Rand::GetHeader( void )
   {
   if ( header )
      return header;

   //-- make the header string from the distName and parameter? strings

   UINT strlength = strlen( distName );
   strlength += strlen( parameterA );
   strlength += strlen( parameterB );
   strlength += strlen( parameterC );

   header = new char[ strlength + 1 + 4 ];   // 4= 2 commas & 2 parenthesis
   strcpy( header, distName );
   strcat( header, "(" );
   strcat( header, parameterA );
```

```
   strcat( header, "," );
   strcat( header, parameterB );
   strcat( header, "," );
   strcat( header, parameterC );
   strcat( header, ")" );

   return header;
   }


void Rand::SetName( char  *_name )
   {
   if ( name )
      delete name;

   if ( _name )
      {
      name = new char[ strlen( _name ) + 1 ];
      strcpy( name, _name );
      }
   else
      name = NULL;
   }

void Rand::SetDistName( char  *_distName )
   {
   if ( distName )
      delete distName;

   if ( _distName )
      {
      distName = new char[ strlen( _distName ) + 1 ];
      strcpy( distName, _distName );
      }
   else
      distName = NULL;
   }


void Rand::SetParameterA_Name( char  *_parameterA )
   {
   if ( parameterA )
      delete parameterA;

   if ( _parameterA )
      {
      parameterA = new char[ strlen( _parameterA ) + 1 ];
      strcpy( parameterA, _parameterA );
      }
   else
      parameterA = NULL;
   }


void Rand::SetParameterB_Name( char  *_parameterB )
   {
   if ( parameterB )
      delete parameterB;

   if ( _parameterB )
      {
      parameterB = new char[ strlen( _parameterB ) + 1 ];
      strcpy( parameterB, _parameterB );
      }
   else
      parameterB = NULL;
   }


void Rand::SetParameterC_Name( char  *_parameterC )
   {
   if ( parameterC )
      delete parameterC;

   if ( _parameterC )
      {
      parameterC = new char[ strlen( _parameterC ) + 1 ];
      strcpy( parameterC, _parameterC );
      }
   else
      parameterC = NULL;
   }


//-- get random number from generator --//
double Rand::GetUnif01( void )
   {
   //-- return a U( 0, 1 ) --//
   seed = fmod( ( (double) multiplier * (double) seed ), (double) LONG_MAX );
   return( (double) seed / (double) LONG_MAX );
   }

void Rand::SetDefaultSeed( void )
   {
   seed = GetDefaultSeed();

   initialSeed = seed;      // the user still has a chance to
                            // reset the initial seed by calling
                            // SetSeed( long _seed );

   return;
   }

long Rand::GetDefaultSeed( void )
   {
   //-- since defaultSeed is static, it is shared by all classes and
   //   hence all instances

   if ( defaultSeed >= 2145000000 )       // 2 billion 145 million
      defaultSeed = 0;

   defaultSeed+=1000000;                  // 1 million

   return defaultSeed;
   }


void Rand::SetSeed( long _seed )
   {
   if ( _seed <= 0 || _seed >= LONG_MAX )
```

```
      SetDefaultSeed();
   else
      seed = _seed;

   SetInitialSeed( seed );

   return;
   }


void Rand::SetInitialSeed( long _seed )
   {
   static UINT initialSeedSet = 0;      // static to an instance of the class

   if ( initialSeedSet )                // only set once per instance of class
      return;

   initialSeedSet = 1;

   initialSeed = _seed;

   return;
   }


void Rand::SelectMultiplierType( UINT _multiplier )
   {

   switch( _multiplier )
      {
      case 2:
         SelectMultiplierType2();
         break;

      case 3:
         SelectMultiplierType3();
         break;

      case 4:
         SelectMultiplierType4();
         break;

      case 5:
         SelectMultiplierType5();
         break;

      case 6:
         SelectMultiplierType6();
         break;

      case 7:
         SelectMultiplierType7();
         break;

      case 8:
         SelectMultiplierType8();
         break;

      default:
         SelectMultiplierType1();

      } // end of switch

return;
}
```

```
Print date: 8/17/1992     Print time:  8:28
//-----------------------------------------------------------------
// PROGRAM: randbern.hpp
//
// PURPOSE: Generates bernoulli random numbers
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   Note: Formula derived from Law & Keltons ( see reference ).
      1. Generate U( 0, 1 )
      2. If U <= probability, return X = 1. Otherwise return X = 0;
      [ Law & Kelton, 1991 ]

   Discrete probability distributions are appropriate models of random
phenomena only if the random variables are measured by counting.

Parameters: P - probability of success where outcome x=0 indicates
            failure or outcome x=1 indicates success.

Range:      [ 0 or 1 ]

Mean:       probability

Variance:   probability * ( 1 - probability )

Type:       Probability is the probability of success during a particular
            sequence of trials.

   "Bernoulli trials are chance experiments in which the outcome of
each trial is expressed as either a success or failure. The P[ success ] = p,
where p is a constant for a particular sequence of trials." [ Bulgren, 1982 ]

References: Discrete System Simulation by William G. Bulgren 1982, pg. 175.

            Simulation Modeling & Analysis by Law and Kelton, 1991,
            pg., 496-497.
-------------------------------------------------------------------
*/

#if !defined _RANDBERN_HPP
#define      _RANDBERN_HPP 1

//------------------------------ Includes -----------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandBernoulli : public Rand
   {

   private:

      double probability;

   public:

      //-- constructors --//

      RandBernoulli( double _probability, char *_name = "-" );
      RandBernoulli( double _probability, UINT _stream, long _seed,
           char *_name = "-" );

      //-- destructor --//
      virtual ~RandBernoulli( void ) { }

      //-- returns a bernoulli random variable --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:35
//---------------------------------------------------------------------
//  PROGRAM: randbern.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDBERN_HPP )
#include "randbern.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandBernoulli::RandBernoulli( double _probability, char *_name )
   : Rand( 1, _name ),
     probability( fabs( _probability ) )
   {
   SetDistName( "Bern" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", probability );
   SetParameterA_Name( temp );
   }


RandBernoulli::RandBernoulli( double _probability, UINT _stream,
      long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     probability( fabs( _probability ) )
   {
   SetDistName( "Bern" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", probability );
   SetParameterA_Name( temp );
   }


double RandBernoulli::RandValue( void )
   {
   AddOneObservation();

   return( ( GetUnif01() <= probability ) ? 1 : 0 );
   }
```

```
Print date: 8/17/1992     Print time:  8:30
//-------------------------------------------------------------------
// PROGRAM: randbeta.hpp
//
// PURPOSE: Generates beta random numbers
//
// NOTE: A beta( 1, 1 ) is a U( 0, 1 ). The beta is bound between
//       0 and 1 unless translated to a different range.
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   Generate Y1 ~ gamma( shape1, 1 ) and Y2 ~ gamma( shape2, 1 ) independently.
   Return X = Y1 / ( Y1 + Y2 )


Parameters: both shape parameters are real and > 0.

Range:      [ lowerBound, upperBound ]

Mean:       "of the Beta( 0, 1 ) variate" is: shape1 / ( shape1 + shape2 )

Var:        "of the Beta( 0, 1 ) variate" is: ( shape1 * shape2 ) /
            ( ( shape1 + shape2 )^2 * ( shape1 + shape2 + 1 ) )


  Used as a rough model in the absence of data. The beta is able to
take on a wide variety of shapes. Although the range of the beta is
between ( 0, 1 ), it can be scaled to any upperBound - lowerBound
range by using the simple formula X = lowerBound + ( upperBound -
lowerBound ) * X', where X' is the beta( 0, 1 ) variate.

References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 338-339, 479-484, and 492-493.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 170-171.
---------------------------------------------------------------------
*/

#if !defined _RANDBETA_HPP
#define      _RANDBETA_HPP 1

//---------------------------- Includes --------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

//---------------------- Forward references ----------------------//

class RandGamma;


class RandBeta : public Rand
  {

  protected:

    double shape1;
    double shape2;
    double upperBound;
    double lowerBound;

    RandGamma *pRNG1;          // these must be independent!
    RandGamma *pRNG2;

    //-- initialize the random number generators --//
    void initialize( double _shape1, double _shape2,
         UINT _stream, long _seed );

  public:

    //-- constructors --//
    RandBeta( double _shape1, double _shape2, double _lowerBound,
         double _upperBound, char *_name = "-" );
    RandBeta( double _shape1, double _shape2, double _lowerBound,
         double _upperBound, long _seed, char *_name = "-" );
    RandBeta( double _shape1, double _shape2, double _lowerBound,
         double _upperBound, UINT _stream, long _seed,
         char *_name = "-" );

    //-- destructor --//
    virtual ~RandBeta( void );

    //-- return an beta random number --//
    virtual double RandValue( void );

  };

#endif
```

```
Print date: 8/17/1992     Print time:  8:33
//------------------------------------------------------------------
//  PROGRAM: randbeta.cpp
//------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDBETA_HPP )
#include "randbeta.hpp"
#endif

#if !defined ( _RANDGAMA_HPP )
#include "randgama.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}

//-- constructors --//

RandBeta::RandBeta( double _shape1, double _shape2, double _lowerBound,
      double _upperBound, char *_name )
   : Rand( 1, _name ),
     shape1( fabs( _shape1 ) ),
     shape2( fabs( _shape2 ) ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound ),
     pRNG1( NULL ),
     pRNG2( NULL )

   {
   SetDistName( "Beta" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape1 );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", shape2 );
   SetParameterB_Name( temp );

   Initialize( shape1, shape2, 1, 1 );
   }


RandBeta::RandBeta( double _shape1, double _shape2, double _lowerBound,
      double _upperBound, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     shape1( fabs( _shape1 ) ),
     shape2( fabs( _shape2 ) ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound ),
     pRNG1( NULL ),
     pRNG2( NULL )

   {
   SetDistName( "Beta" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape1 );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", shape2 );
   SetParameterB_Name( temp );

   Initialize( shape1, shape2, 1, _seed );
   }


RandBeta::RandBeta( double _shape1, double _shape2, double _lowerBound,
      double _upperBound, UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     shape1( fabs( _shape1 ) ),
     shape2( fabs( _shape2 ) ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound ),
     pRNG1( NULL ),
     pRNG2( NULL )

   {
   SetDistName( "Beta" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape1 );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", shape2 );
   SetParameterB_Name( temp );

   Initialize( shape1, shape2, _stream, _seed );
   }


RandBeta::~RandBeta( void )
   {
   if ( pRNG1 )
      delete pRNG1;

   if ( pRNG2 )
      delete pRNG2;
   }

//-- initialize the random number generators --//
void RandBeta::Initialize( double _shape1, double _shape2, UINT _stream,
      long _seed )
   {
   pRNG1 = new RandGamma( _shape1, 1, _stream, _seed );

   //-- new stream # so this is independent of pRNG1 --//
   pRNG2 = new RandGamma( _shape2, 1, _stream + 1, _seed );

   return;
   }


//-- return a beta random number --//
```

```
double RandBeta::RandValue( void )
   {
   AddOneObservation();

   double X = 0;

   double Y1 = pRNG1->RandValue();      // get a gamma variate
   double Y2 = pRNG2->RandValue();      // get a gamma variate

   X = Y1 / ( Y1 + Y2 );

   return ( lowerBound + ( upperBound - lowerBound ) * X );
   }
```

```
Print date: 8/17/1992     Print time:  8:29
//------------------------------------------------------------------
// PROGRAM: randbino.hpp
//
// PURPOSE: Generates binomial random variate
//
// NOTE:    RandBinomial( successfullTrials, probability )
//------------------------------------------------------------------

/*
--------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   Since the sum of t IID Bernoulli random variables has the
   Binomial distribution, a simple convolution algorithm produces
   the desired distribution.

   Generate Y1, Y2, ..., Yt as IID Bernoulli( prob ) random variates.
   Then return X = Y1 + Y2 + ... + Yt. [ Law and Kelton, 1991 ]
   The random variable X equals the number of successes in t trials.

   Other methods are discussed in Law and Kelton which deal with
   the problem of summing a large number of Bernoulli variates. Also,
   one could consider the normal approximation if the number of
   Bernoulli trials to sum is large. One caveat to the normal
   approximation is the possibility of negative values. For a complete
   discussion see [ Bulgren, 1982 ].

Parameters: successfullTrials( t ) is a positive integer, and probability
            ( p ) is 0 < p < 1.

Range:      ( 0, 1, ..., t )

Mean:       t*p or successfulTrials * probability

Var:        t*p( 1 - p )

   The binomial distribution can be used in many situations but one
example to consider is if we let binomial variable X with parameters
Y = number of Yes's, Z = number of No's, and p probability of a yes,
then N = Y + Z where N becomes the random binomial variate.
   "Number of successes in t independent Bernoulli trials with prob-
ability p of success on each trial; number of "defective" items in a
batch of size t; number of items in a batch ( e.g., a group of
people ) of random size; number of items demanded from an inventory."
[ Law & Kelton, 1991 ]


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 345-346 and 502.

            Discrete System Simulation by William G. Bulgren, 1982,
            pgs., 176-177.
--------------------------------------------------------------------
*/

#if !defined _RANDBINO_HPP
#define      _RANDBINO_HPP 1

//---------------------------- Includes ------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

//------------------------- Forward reference -------------------//

class RandBernoulli;


class RandBinomial : public Rand
   {

   private:

      double probability;              // probability of a success
      ULONG  successfullTrials;

      RandBernoulli *pRNG;             // since we sum Bernoulli's

   public:

      //-- constructors --//
      RandBinomial( ULONG successfullTrials, double _probability,
            char *_name = "-" );

      RandBinomial( ULONG successfullTrials, double _probability,
            UINT _stream, long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandBinomial( void );

      //-- returns a binomial random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:32
//---------------------------------------------------------------
//  PROGRAM: randbino.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDBINO_HPP )
#include "randbino.hpp"
#endif

#if !defined ( _RANDBERN_HPP )
#include "randbern.hpp"
#endif

extern "C"
{
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandBinomial::RandBinomial( ULONG _successfullTrials, double _probability,
      char *_name )
   : Rand( 1, _name ),
     probability( _probability ),
     successfullTrials( _successfullTrials )
   {
   SetDistName( "Binomial" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", successfullTrials );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", probability );
   SetParameterB_Name( temp );

   pRNG = new RandBernoulli( probability );
   }


RandBinomial::RandBinomial( ULONG _successfullTrials, double _probability,
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     probability( _probability ),
     successfullTrials( _successfullTrials )
   {
   SetDistName( "Binomial" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", successfullTrials );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", probability );
   SetParameterB_Name( temp );

   pRNG = new RandBernoulli( probability, _stream, _seed );
   }


//-- destructor --//
RandBinomial::~RandBinomial( void )
   {
   if ( pRNG )
      delete pRNG;
   }


//-- returns a binomial variate --//
double RandBinomial::RandValue( void )
   {
   AddOneObservation();

   ULONG success = 0;

   for ( ULONG i = 1; i < successfullTrials; i++ )
     success += pRNG->RandValue();

   return (double) success;
   }
```

```
Print date: 8/17/1992     Print time:  8:30
//---------------------------------------------------------------------
// PROGRAM: randchi.hpp
//
// PURPOSE: Generates chi-square distributed random numbers
//---------------------------------------------------------------------

/*
---------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   We can exploit the gamma function to generate Chi^2 variates by
   setting gamma( k / 2, 2 ), where k is the degrees of freedom of
   the Chi^2 variate.


Parameters: degreesOfFreedom is an integer >= 1.

Range:      [ 0, +infinity ]

Mean:       degreesOfFreedom

Var:        2 * degreesOfFreedom


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 332, 336, 383, 384, 560, 484-485.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 170.

            Statistical Distributions by Hastings and Peacock, 1975,
            pgs., 46-51.
---------------------------------------------------------------------
*/

#if !defined _RANDCHI_HPP
#define      _RANDCHI_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


//--------------------------- Forward reference -----------------//

class RandGamma;


class RandChiSquare : public Rand
   {
   private:

      RandGamma *pRNG;              // Chi is gamma with shape = ( alpha/2 )
                                    // and scale = 2

      UINT degreesOfFreedom;        // k df

   public:

      //-- constructors --//
      RandChiSquare( UINT _degreesOfFreedom, long _seed, char *_name = "-" );
      RandChiSquare( UINT _degreesOfFreedom, UINT _stream, long _seed,
            char *_name = "-" );

      //-- destructor --//
      virtual ~RandChiSquare( void );

      //-- returns a chi-square random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:32
//---------------------------------------------------------------
//  PROGRAM: randchi.cpp
//---------------------------------------------------------------

#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _RANDCHI_HPP )
#include "randchi.hpp"
#endif

#if !defined ( _RANDGAMA_HPP )
#include "randgama.hpp"
#endif

extern "C"
{
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandChiSquare::RandChiSquare( UINT _degreesOfFreedom, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     pRNG( NULL ),
     degreesOfFreedom( _degreesOfFreedom )
   {
   SetDistName( "ChiSquare" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", degreesOfFreedom );
   SetParameterA_Name( temp );

   //-- run the constructor for the gamma distribution --//
   pRNG = new RandGamma( (double) degreesOfFreedom / 2.0, 2.0, _seed );
   }

RandChiSquare::RandChiSquare( UINT _degreesOfFreedom, UINT _stream,
      long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     pRNG( NULL ),
     degreesOfFreedom( _degreesOfFreedom )
   {
   SetDistName( "ChiSquare" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", degreesOfFreedom );
   SetParameterA_Name( temp );

   //-- run the constructor for the gamma distribution --//
   pRNG = new RandGamma( (double) degreesOfFreedom / 2.0, 2.0, _stream, _seed );
   }

RandChiSquare::~RandChiSquare( void )
   {
   if ( pRNG )
      delete pRNG;
   }

double RandChiSquare::RandValue( void )
   {
   AddOneObservation();

   return( pRNG->RandValue() );      // get a gamma variate
   }
```

```
Print date: 8/17/1992     Print time:  8:27
//------------------------------------------------------------------
// PROGRAM: randduni.hpp
//
// PURPOSE: Generates discrete uniform ( 0, 1 ) random numbers
//          or discrete uniform ( lowerBound, upperBound ) random
//          numbers where lowerBound < upperBound
//------------------------------------------------------------------

/*
--------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   Generate U ~ U( 0, 1 )
   Return X = lowerBound + <[ ( upperBound - lowerBound + 1 ) * U ]>
   where we take the largest integer between the <[]>.

Parameters: lowerBound and upperBound are integers.

Range:      ( i, i + 1, ...., j )

Mean:       ( lowerBound + upperBound ) / 2

Var:        ( ( upperBound - lowerBound + 1 )^2 - 1 ) / 12

   "Random occurrences with several possible outcomes, each of which
is equally likely; used as a "first" model for a quantity that is
varying among the integers i through j but about which little else is
known." [ Law & Kelton, 1991 ]


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 344-345 and 497.
--------------------------------------------------------------------
*/

#if !defined _RANDDUNI_HPP
#define      _RANDDUNI_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandDiscreteUniform : public Rand
   {

   private:

      //-- the discrete bounds for the uniform variate --//
      long lowerBound;
      long upperBound;

      long largestInt;           // algorithm coefficient

   public:

      //-- constructors --//
      RandDiscreteUniform( char *_name = "-" );
      RandDiscreteUniform( UINT _stream, long _seed, char *_name = "-" );
      RandDiscreteUniform( long _lowerBound, long _upperBound, long _seed,
            char *_name = "-" );
      RandDiscreteUniform( long _lowerBound, long _upperBound, \
            UINT _stream, long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandDiscreteUniform( void ) { }

      //-- return a discrete uniform number --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:35
//---------------------------------------------------------------
//  PROGRAM: randduni.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDDUNI_HPP )
#include "randduni.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandDiscreteUniform::RandDiscreteUniform( char *_name )
   : Rand( 1, _name ),
     lowerBound( 0 ),
     upperBound( 1 ),
     largestInt( 2 )
   {
   SetDistName( "DiscUniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", 0 );
   SetParameterA_Name( temp );
   sprintf( temp, "%u", 1 );
   SetParameterB_Name( temp );
   }


RandDiscreteUniform::RandDiscreteUniform( UINT _stream, long _seed, char *_name
   : Rand( _stream, _seed, _name ),
     lowerBound( 0 ),
     upperBound( 1 ),
     largestInt( 2 )
   {
   SetDistName( "DiscUniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", 0 );
   SetParameterA_Name( temp );
   sprintf( temp, "%u", 1 );
   SetParameterB_Name( temp );
   }

RandDiscreteUniform::RandDiscreteUniform( long _lowerBound, long _upperBound,
      long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound ),
     largestInt( upperBound - lowerBound + 1 )
   {
   SetDistName( "DiscUniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%Lu", upperBound );
   SetParameterB_Name( temp );
   }

RandDiscreteUniform::RandDiscreteUniform( long _lowerBound, long _upperBound,
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound ),
     largestInt( upperBound - lowerBound + 1 )
   {
   SetDistName( "DiscUniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%Lu", upperBound );
   SetParameterB_Name( temp );
   }



double RandDiscreteUniform::RandValue( void )
   {
   AddOneObservation();

   long largestInteger = (long) floor( (double) largestInt * GetUnif01() );

   return ( (double) ( lowerBound + largestInteger ) );
   }
```

```
Print date: 8/17/1992     Print time:  8:27
//-----------------------------------------------------------------
// PROGRAM: randerla.hpp
//
// PURPOSE: Generates erlang distributed random numbers
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   Note: Formula derived from Law & Keltons ( see reference ), incorrect
   formula. "If X is an m-Erlang random variable with mean Beta, we can
   write X = Y1 + Y2 + ... + Ym, where Yi's are IID exponential random
   variables, each with mean ( Beta / m )." [ Law and Kelton, 1991 ]

   It should be noted that the m-Erlang is a special case of the gamma
distribution ( with shape parameter alpha equal to the integer m ).

Parameters: If X1, X2, ..., Xm are independent exponential samples, then
            the sum of these m samples has an Erlang-m distribution. The
            mean Beta of the exponential distribution and the number of
            exponential samples m are parameters of the distribution.

Range:      [ 0, +infinity ]

Mean:       m * Beta

Variance:   m * Beta^2

Type:       Beta is positive real and m is a positive integer.

   "Erlang distribution is used in situations in which an activity occurs
in phases and each phase has an exponential distribution. For large m the
Erlang approaches the normal distribution. The Erlang distribution is often
used to represent the time required to complete a task. The Erlang
distribution is a special case of the gamma distribution in which the shape
parameter, Alpha, is an integer". [ Pegden et al, 1990 ]

References: Simulation Modeling & Analysis by Averill M. Law and W. David
            Kelton, 1991, pg., 486.

            Introduction to Simulation using SIMAN by C. Dennis Pegden,
            Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 562.
-------------------------------------------------------------------
*/

#if !defined _RANDERLA_HPP
#define      _RANDERLA_HPP 1

//--------------------------- Includes --------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandErlang : public Rand
   {

   private:

      UINT shape;             // number of samples
      double scale;           // exponential mean

      double a;               // algorithm coefficients, See Law & Kelton

   public:

      //-- constructors --//
      RandErlang( UINT _shape, double _scale, char *_name = "-" );
      RandErlang( UINT _shape, double _scale, UINT _stream,
            long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandErlang( void ) { }

      //-- returns an erlang random number --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:35
//-------------------------------------------------------------------
//  PROGRAM: randerla.cpp
//-------------------------------------------------------------------

#pragma hdrstop

//--------------------------- Includes --------------------------//

#if !defined ( _RANDERLA_HPP )
#include "randerla.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandErlang::RandErlang( UINT _shape, double _scale, char *_name )
   : Rand( 1, _name ),
     shape( _shape ),
     scale( fabs( _scale ) ),
     a( - ( scale / (double) _shape ) )

   {
   SetDistName( "Erlang" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );
   }

RandErlang::RandErlang( UINT _shape, double _scale,
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     shape( _shape ),
     scale( fabs( _scale ) ),
     a( - ( scale / (double) _shape ) )

   {
   SetDistName( "Erlang" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );
   }
```

```
double RandErlang::RandValue( void )
   {
   AddOneObservation();

   double multFactor = GetUnif01();

   for( UINT i = 1; i < shape; i++ )
      {
      //-- get more U(0,1) numbers --//
      multFactor = multFactor * GetUnif01();
      }

   return( - 1/scale * log( multFactor ) );
   }
```

```
Print date: 8/17/1992     Print time:  8:29
//-----------------------------------------------------------------
// PROGRAM: randexp.hpp
//
// PURPOSE: Generates exponential random numbers
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   Note: Inverse transform method where X = - mean * ( ln( 1 - U( 0,1 ) ) )
   Which is the same as X = -mean * ( ln( U( 0,1 ) ); [ Bulgren 1982 ]
   Beta = 1 / rate term lamda [ Thesen and Travis, 1992 ]

Parameters: mean is the rate parameter and is > 0.

Range:      [ 0, +infinity ]

Mean:       Beta ( mean )

Variance:   Beta ^ 2

   "This distribution is often used to model random arrival and breakdown
processes, but it is generally inappropriate for modeling process delay
times. The exponential distribution is used in situations in which the
random quantities satisfy the memoryless property." [ Pegden et al, 1990 ]


Reference: Discrete System Simulation by William G. Bulgren 1982, pg. 168.

           Simulation for Decision Making by Arne Thesen and
           Laurel E. Travis, 1992, pg. 24.

           Introduction to Simulation Using SIMAN by C. Dennis Pegden,
           Robert E. Shannon, and Randall P. Sadowski, 1990, pg. 561.
-------------------------------------------------------------------
*/

#if !defined _RANDEXP_HPP
#define      _RANDEXP_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandExponential : public Rand
   {

   private:

      double mean;            // rate parameter

   public:

      //-- constructors --//
      RandExponential( double _mean, char *_name = "-" );
      RandExponential( double _mean, long _seed, char *_name = "-" );
      RandExponential( double _mean, UINT _stream, long _seed,
                       char *_name = "-" );

      //-- destructor --//
      virtual ~RandExponential( void ) { }

      //-- return an exponential random number --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:35
//---------------------------------------------------------------
//  PROGRAM: randexp.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes -------------------------//

#if !defined ( _RANDEXP_HPP )
#include "randexp.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandExponential::RandExponential( double _mean, char *_name )
   : Rand( 1, _name ),
     mean( fabs( _mean ) )
   {
   SetDistName( "Expon" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", mean );
   SetParameterA_Name( temp );
   }

RandExponential::RandExponential( double _mean, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     mean( fabs( _mean ) )
   {
   SetDistName( "Expon" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", mean );
   SetParameterA_Name( temp );
   }

RandExponential::RandExponential( double _mean, UINT _stream,
   long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     mean( fabs( _mean ) )
   {
   SetDistName( "Expon" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", mean );
   SetParameterA_Name( temp );
   }


//-- return an exponential random number --//
double RandExponential::RandValue( void )
   {
   AddOneObservation();

   return ( -mean * log( GetUnif01() ) );
   }
```

```
Print date: 8/17/1992     Print time:  8:29
//-------------------------------------------------------------------
// PROGRAM: randgama.hpp
//
// PURPOSE: Generates gamma random numbers
//
// NOTE: If shape( alpha ) = 1 use exponential, if shape is positive
//       integer use Erlang. To generate Chi-square with "v" degrees
//       of freedom, use shape = ( v/2 ) and scale( beta ) = 2.
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   There are three cases to consider and all depend on shape ( alpha ).
   case 1: shape < 1    Aherns & Dieter ( 1974 )
   case 2: shape = 1    Exponential with mean scale ( beta )
   case 3: shape > 1    Cheng ( 1977 )


Parameters: both shape parameters are real and > 0.

Range:      [ 0, +infinity ]

Mean:       shape * scale

Var:        shape * scale * scale


   "Time to complete some task, e.g., customer service or machine repair"
   [ Law & Kelton, 1991 ]


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 331-333, 487-490, and 518.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 169-170.
---------------------------------------------------------------------
*/

#if !defined _RANDGAMA_HPP
#define      _RANDGAMA_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

#include <math.h>


//---------------------- Forward references --------------------//

class RandGamma;
class RandExponential;
class RandUniform;
```

```
//---------------------------- Typedefs -------------------------//

//-- gamma random value generation method --//
//   this points to one of three possible functions
typedef double (RandGamma::*RANDPROC)( void );


class RandGamma : public Rand
   {

   private:

      double shape;           // alpha parameter
      double scale;           // beta  parameter

      //-- Cheng (1977) for alpha > 1 --//
      double a;               // a = 1 / sqrt( 2 * alpha - 1 )
      double b;               // b = alpha - ln 4
      double q;               // q = alpha + 1 / a
      double phi;             // phi = 4.5
      double d;               // d = 1 + ln( phi )

      //-- Ahrens and Dieter (1974) is acceptance - rejection technique --//
                              // b = ( e + alpha ) / e

      //-- initialize the gamma generator --//
      void Initialize( UINT _stream, long _seed );
      void InitializeGreaterThanOne( UINT _stream, long _seed );
      void InitializeLessThanOne( UINT _stream, long _seed );
      void InitializeEqualOne( UINT _stream, long _seed );

      //-- get a random gamma value based on alpha --//
      double GreaterThanOne( void );
      double LessThanOne( void );
      double EqualOne( void );

      //-- pointer to the random generator method --//
      RANDPROC pRandFn;


   protected:

      RandExponential *pRNG_Exp;    // for alpha == 1

      //-- these will use different streams --//
      RandUniform     *pRNG_One;    // otherwise
      RandUniform     *pRNG_Two;    // otherwise

   public:

      //-- constructors --//
      RandGamma( double _shape, double _scale, char *_name = "-" );
      RandGamma( double _shape, double _scale, long _seed, char *_name = "-" )
      RandGamma( double _shape, double _scale, UINT _stream, long _seed,
            char *_name = "-" );

      //-- destructor --//
      virtual ~RandGamma( void );

      //-- return an gamma random number --//
```

```
        virtual double RandValue( void );
    };

#endif
```

```
Print date: 8/17/1992     Print time:  8:37
//---------------------------------------------------------------
//  PROGRAM: randgama.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDGAMA_HPP )
#include "randgama.hpp"
#endif

#if !defined ( _RANDEXP_HPP )
#include "randexp.hpp"
#endif

#if !defined ( _RANDUNIF_HPP )
#include "randunif.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandGamma::RandGamma( double _shape, double _scale, char *_name )
   : Rand( 1, _name ),
     shape( fabs( _shape ) ),
     scale( fabs( _scale ) ),
     a( 0 ),
     b( 0 ),
     q( 0 ),
     phi( 0 ),
     d( 0 ),

     pRNG_Exp( NULL ),
     pRNG_One( NULL ),
     pRNG_Two( NULL )
   {
   SetDistName( "Gamma" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );

   Initialize( 1, 0 );     // default stream, default seed
   }

RandGamma::RandGamma( double _shape, double _scale, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     shape( fabs( _shape ) ),
     scale( fabs( _scale ) ),
     a( 0 ),
     b( 0 ),
     q( 0 ),
     phi( 0 ),
     d( 0 ),

     pRNG_Exp( NULL ),
     pRNG_One( NULL ),
     pRNG_Two( NULL )
   {
   SetDistName( "Gamma" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );

   Initialize( 1, _seed );    // default stream
   }

RandGamma::RandGamma( double _shape, double _scale, UINT _stream,
   long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     shape( fabs( _shape ) ),
     scale( fabs( _scale ) ),
     a( 0 ),
     b( 0 ),
     q( 0 ),
     phi( 0 ),
     d( 0 ),

     pRNG_Exp( NULL ),
     pRNG_One( NULL ),
     pRNG_Two( NULL )
   {
   SetDistName( "Gamma" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );

   Initialize( _stream, _seed );
   }


RandGamma::~RandGamma( void )
   {
   if ( pRNG_Exp )
      delete pRNG_Exp;

   if ( pRNG_One )
      delete pRNG_One;
```

```
    if ( pRNG_Two )
       delete pRNG_Two;
    }


void RandGamma::Initialize( UINT _stream, long _seed )
    {
    if ( shape < 1 )
       InitializeLessThanOne( _stream, _seed );

    else if ( shape == 1 )
       InitializeEqualOne( _stream, _seed );

    else
       InitializeGreaterThanOne( _stream, _seed );

    return;
    }


//-- return an gamma random variable --//
double RandGamma::RandValue( void )
    {
    AddOneObservation();

    //-- use the appropriate method based on initialization --//
    double gammaOne = (this->*pRandFn)();  // returns gamma( shape, 1 )

    return ( scale * gammaOne );
    }


void RandGamma::InitializeGreaterThanOne( UINT _stream, long _seed )
    {
    //-- shape( shape ) is greater than one --//

    a = 1 / sqrt( 2 * shape - 1 );
    b = shape - log( 4 );
    q = shape + 1 / a;
    phi = 4.5;
    d = 1 + log( phi );

    //-- set the function pointer to the correct method --//
    pRandFn = (RANDPROC) &RandGamma::GreaterThanOne;

    //-- new the Uniform generators --//
    pRNG_One = new RandUniform( _stream, _seed );
    pRNG_Two = new RandUniform( _stream + 1, _seed );

    return;
    }

double RandGamma::GreaterThanOne( void )
    {
    //-- shape( shape ) is greater than one --//

    while( 1 )
       {
       double U_One = pRNG_One->RandValue();

       double U_Two = pRNG_Two->RandValue();

       double V = a * log( U_One / ( 1 - U_One ) );
       double Y = shape * exp( V );
       double Z = U_One * U_One * U_Two;
       double W = b + q * V - Y;

       double temp1 = W + d - phi * Z;
       double temp2 = log( Z );

       if ( temp1 >= 0 || W >= temp2 )
          return Y;

       } // end of WHILE

    }

void RandGamma::InitializeLessThanOne( UINT _stream, long _seed )
    {
    //-- shape( alpha ) is < 1 --//

    b = ( exp( 1 ) + shape ) / exp( 1 );

    //-- set the function pointer to the correct method --//
    pRandFn = (RANDPROC) &RandGamma::LessThanOne;

    //-- new the Uniform generators --//
    pRNG_One = new RandUniform( _stream, _seed );
    pRNG_Two = new RandUniform( _stream + 1, _seed );

    return;
    }

double RandGamma::LessThanOne( void )
    {
    //-- shape( alpha ) is less than one --//

    while( 1 )
       {
       double U_One = pRNG_One->RandValue();
       double P = b * U_One;

       if ( P <= 1 )
          {
          double Y = pow( P, ( 1/shape ) );
          double U_Two = pRNG_Two->RandValue();

          if ( U_Two <= exp( -Y ) )
             return Y;
          }

       else
          {
          double Y = - log( ( b - P ) / shape );
          double U_Two = pRNG_Two->RandValue();

          if ( U_Two <= pow( Y, ( shape - 1 ) ) )
             return Y;
          }
```

```
    } // end of WHILE

  }


void RandGamma::InitializeEqualOne( UINT _stream, long _seed )
  {
  //-- set the function pointer to the correct method --//
  pRandFn = (RANDPROC) &RandGamma::EqualOne;

  //-- new the Exponential generator --//
  pRNG_Exp = new RandExponential( 1, _stream, _seed );

  return;
  }

double RandGamma::EqualOne( void )
  {
  //-- shape( alpha ) is == one --//
  //   same as exponential       //

  return ( pRNG_Exp->RandValue() );
  }
```

```
Print date: 8/17/1992     Print time:  8:25
//------------------------------------------------------------------
// PROGRAM: randgeo.hpp
//
// PURPOSE: Generates geometric random numbers
//
// NOTE: RandGeometric( probability )
//------------------------------------------------------------------

/*
--------------------------------------------------------------------
   If Y1, Y2, ..., is a sequence of independent Bernoulli trials,
   then X = the number of failures before the first success.

Parameters: probability ( p ) is 0 < p < 1.

Range:      ( 0, 1, ... )

Mean:       ( 1 - p ) / p

Var:        ( 1 - p ) / ( p * p )

   "Number of failures before the first success in a sequence of in-
dependent Bernoulli trials with probability p of success on each trial;
number of items inspected before encountering the first defective item;
number of items in a batch of random size; number of items demanded from
an inventory". [ Law & Kelton, 1991 ]


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 347 and 502.

            Discrete System Simulation by William G. Bulgren, 1982,
            pgs., 175-176.
--------------------------------------------------------------------
*/

#if !defined _RANDGEO_HPP
#define      _RANDGEO_HPP 1

//----------------------------- Includes ------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

//------------------------- Forward reference -------------------//

class RandBernoulli;


class RandGeometric : public Rand
   {

   private:

      double probability;           // probability of success at each trial

      RandBernoulli *pRNG;          // since we sum Bernoulli trials

   public:

      //-- constructors --//
      RandGeometric( double _probability,  char *_name = "-" );

      RandGeometric( double _probability, UINT _stream,
            long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandGeometric( void );

      //-- returns a geometric random variable --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:32
//---------------------------------------------------------------
//  PROGRAM: randgeo.cpp
//---------------------------------------------------------------

#pragma hdrstop

//------------------------- Includes -------------------------//

#if !defined ( _RANDGEO_HPP )
#include "randgeo.hpp"
#endif

#if !defined ( _RANDBERN_HPP )
#include "randbern.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandGeometric::RandGeometric( double _probability, char *_name )
   : Rand( 1, _name ),
     probability( fabs( _probability ) )
   {
   SetDistName( "Geometric" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", probability );
   SetParameterA_Name( temp );

   pRNG = new RandBernoulli( probability );
   }


RandGeometric::RandGeometric( double _probability, UINT _stream,
      long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     probability( fabs( _probability ) )
   {
   SetDistName( "Geometric" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", probability );
   SetParameterA_Name( temp );

   pRNG = new RandBernoulli( probability, _stream, _seed );
   }

//-- destructor --//

RandGeometric::~RandGeometric( void )
   {
   if ( pRNG )
      delete pRNG;
   }


//-- returns a geometric variate --//
double RandGeometric::RandValue( void )
   {
   AddOneObservation();

   //-- number of failures before the 1st success --//
   long numTrials = -1;
   UINT bernoulliResult = 0;

   while( bernoulliResult == 0 )
      {
      bernoulliResult = (UINT) pRNG->RandValue();

      numTrials++;
      }

   return (double) numTrials;
   }
```

```
Print date: 8/17/1992     Print time:  8:29
//-------------------------------------------------------------------
// PROGRAM: randln.hpp
//
// PURPOSE: Generates lognormal distributed random numbers
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference
   "A special property of the lognormal distribution, namely,
   that if Y ~ N( mean, var ) then exp( Y ) ~ LN( mean, var ),
   is used to obtain the following algorithm:
      1. Generate Y ~ N( mean, var )
      2. Set X = exp( Y )

      where meanLogNorm = mean of lognormal distribution
            varLogNorm  = var of lognormal distribution

      mean = ln[ meanLogNorm ^ 2 / sqrt( varLogNorm + meanLogNorm ^ 2 ) ]
      var  = ln[ ( varLogNorm + meanLogNorm ^ 2 ) / meanLogNorm ^ 2 ]

      Then get Y ~ N( mean, var )
      and the X = exp( Y ) [ Law and Kelton, 1982 ]

Parameters: The mean and the standard deviation specified as positive
            real numbers.

Range:      [ 0, +infinity ]

Mean:       meanLogNorm
Var:        varLogNorm

   "The lognormal distribution is used in situations in which the
quantity is the product of a large number of random quantities. It is
also used to represent task times that have a non-symmetric distribution".
[ Pegden et al, 1990 ]

Reference: Simulation Modeling & Analysis by Law and Kelton, 1982,
           pg., 259-260.

           Introduction to Simulation Using SIMAN by C. Dennis Pegden,
           Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 562.
---------------------------------------------------------------------
*/

#if !defined _RANDLN_HPP
#define      _RANDLN_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


//------------------------- Forward reference -------------------//

class RandNormal;

class RandLogNormal : public Rand
   {

   private:

      RandNormal *pRNG;              // RandNormal object ptr

      //-- parameters --//
      double meanLogNorm;            // Log normal mean
      double stdLogNorm;             // Log normal standard deviation
      double varLogNorm;             // Log normal variance

      double mean;                   // Normal mean
      double stdev;                  // Normal standard deviation
      double variance;               // Normal variance

      void SetNormalMean( void );       // create the normal mean
      void SetNormalVariance( void );   // create the normal variance

   public:

      //-- constructor --//
      RandLogNormal( double meanLogNorm, double stdLogNorm,
            long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandLogNormal( void );

      //-- returns a log normal random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:33
//---------------------------------------------------------------------
//  PROGRAM: randln.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDLN_HPP )
#include "randln.hpp"
#endif

#if !defined ( _RANDNORM_HPP )
#include "randnorm.hpp"
#endif


extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructor --//
RandLogNormal::RandLogNormal( double _meanLogNorm, double _stdLogNorm,
      long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     meanLogNorm( fabs( _meanLogNorm ) ),
     stdLogNorm( fabs( _stdLogNorm ) ),
     varLogNorm( stdLogNorm * stdLogNorm )
   {
   SetDistName( "LogNormal" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", meanLogNorm );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", stdLogNorm );
   SetParameterB_Name( temp );

   SetNormalMean();
   SetNormalVariance();

   //-- run this constructor after mean and var have been set --//
   pRNG = new RandNormal( mean, stdev, _seed );
   }


RandLogNormal::~RandLogNormal( void )
   {
   if ( pRNG )
      delete pRNG;
   }


double RandLogNormal::RandValue( void )
   {
   AddOneObservation();

   double Y = pRNG->RandValue();
   return( exp( Y ) );
   }


void RandLogNormal::SetNormalMean( void )
   {
   mean = log( ( meanLogNorm*meanLogNorm /
         sqrt( varLogNorm + meanLogNorm*meanLogNorm ) ) );
   }


void RandLogNormal::SetNormalVariance( void )
   {
   variance = log( ( varLogNorm + meanLogNorm*meanLogNorm ) /
         ( meanLogNorm*meanLogNorm ) );

   stdev = sqrt( variance );
   }
```

```
Print date: 8/17/1992     Print time: 19:58
 //-----------------------------------------------------------------
// PROGRAM: randnbin.hpp
//
// PURPOSE: Generates negative binomial random numbers
//
// NOTE:    Number of failures before the sth success in a sequence of
//          independent bernoulli trials with probability p of success
//          on each trial.
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   Note: Formula derived from Law & Kelton, see reference.
   Generate Y1, Y2, ..., Ys as IID geometric( prob ) variates.
   Return X = Y1 + Y2 + ... + Ys.

   For large sequences of numbers to sum, one should consider the
   consider the formula from Fishman which utilizes the relationship
   between the negative binomial and the Gamma & Poisson distributions.
   [ Fishman, 1973 ]

Parameters: numSuccesses is a positive integer, probability is between
            ( 0, 1 ).

Range:      ( 0, 1, ... )

Mean:       numSuccesses( 1 - probability ) / probability

Var:        numSuccesses( 1 - probability ) / ( probability * probability )


   "Number of failures before the sth success in a sequence of independent
Bernoulli trials with probability p of success on each trial; number of
good items inspected before encountering the sth defective item; number
of items in a batch of random size; number of items demanded from an
inventory." [ Law & Kelton, 1991 ]


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 348-349 and 502-503.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 176.

            Concepts and Methods in Discrete Event Digital Simulation
            by George S. FishMan, 1973, pgs., 226-227.
-------------------------------------------------------------------
*/

#if !defined _RANDNBIN_HPP
#define      _RANDNBIN_HPP 1

//------------------------------ Includes ------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif

//------------------------- Forward reference --------------------//

class RandGeometric;


class RandNegBinomial : public Rand
  {

  private:

     double probability;
     ULONG  numSuccesses;

     RandGeometric *pRNG;

  public:

     //-- constructors --//
     RandNegBinomial( ULONG numSuccesses, double _probability,
           char *_name = "-" );
     RandNegBinomial( ULONG numSuccesses, double _probability,
          UINT _stream, long _seed, char *_name = "-" );

     //-- destructor --//
     virtual ~RandNegBinomial( void );

     //-- returns a bernoulli random variable --//
     virtual double RandValue( void );

  };


#endif
```

```
Print date: 8/17/1992     Print time:  8:34
//---------------------------------------------------------------
//  PROGRAM: randnbin.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDNBIN_HPP )
#include "randnbin.hpp"
#endif

#if !defined ( _RANDGEO_HPP )
#include "randgeo.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandNegBinomial::RandNegBinomial( ULONG _numSuccesses, double _probability,
      char *_name )
   : Rand( 1, _name ),
     numSuccesses( _numSuccesses ),
     probability( fabs( _probability ) )
   {
   SetDistName( "NegBinom" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", numSuccesses );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", probability );
   SetParameterB_Name( temp );

   pRNG = new RandGeometric( probability );
   }


RandNegBinomial::RandNegBinomial( ULONG _numSuccesses, double _probability,
     UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     numSuccesses( _numSuccesses ),
     probability( fabs( _probability ) )
   {
   SetDistName( "NegBinom" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%Lu", numSuccesses );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", probability );
   SetParameterB_Name( temp );

   pRNG = new RandGeometric( probability, _stream, _seed );
   }


//-- destructor --//
RandNegBinomial::~RandNegBinomial( void )
   {
   if ( pRNG )
      delete pRNG;
   }


double RandNegBinomial::RandValue( void )
   {
   AddOneObservation();

   //--number of failures before the sth success in a sequence of
   //  independent bernoulli trials with probability p of success
   //  on each trial

   long geometricResult = 0;

   for ( UINT i = 0; i < numSuccesses; i++ )
      geometricResult += (long) pRNG->RandValue();

   return ( (double) geometricResult );
   }
```

```
Print date: 8/17/1992     Print time:  8:30
//-----------------------------------------------------------------
// PROGRAM: randnorm.hpp
//
// PURPOSE: Generates normal random numbers
//
// NOTE:    Uses polar method
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference
   Given X ~ N( 0, 1 ), we can obtain X' ~ N( mean, variance ) by
   setting X' = mean + variance * X. [ Law and Kelton, 1991 ]

   Normal random variates can also be transformed directly into random
   variates from other distributions, e.g., the lognormal.

   The most famous implementation of normal random variate generation is
the Box-Muller method. An improvement to the Box-Muller method was
described by Marsaglia and Bray ( 1964 ), and is called the polar method.
This normal random variate generator implements the polar method.

   1. Generate U1 and U2 as IID U( 0, 1 ), let Vi = 2Ui - 1 for i = 1, 2,
      and let W = V1 * V1 + V2 * V2.
   2. If W > 1, go back to step 1. Otherwise, let Y = sqrt( ( -2* ln W ) / W )
      and X1 = V1 * Y, and X2 = V2 * Y. The  X1 and X2 are IID N( 0, 1 )
      random variates. [ Law and Kelton, 1991 ]


Parameters: The mean is specified as a real number, and the standard
            deviation is specified as a non-negative real number.

Range:      [ -infinity, +infinity ]

Mean:       mean

Var:        variance

   "The normal distribution is used in situations in which the central
limit theorem applies -- i.e., quantities that are sums of other quantities.
It is also used empirically for many processes that are known to have a
symmetric distribution and for which the mean and standard deviation can
be estimated. Because the theoretical range is from -infinity to +infinity,
the distribution should only be used for processing times when the mean is
at least three standard deviations above 0." [ Pegden et al, 1990 ]

Reference: Simulation Modeling & Analysis by Law and Kelton, 1982,
           pg., 259-260.

           Introduction to Simulation Using SIMAN by C. Dennis Pegden,
           Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 562.
-------------------------------------------------------------------
*/

#if !defined _RANDNORM_HPP
#define      _RANDNORM_HPP 1

//--------------------------- Includes ---------------------------//
#if !defined ( _RANDUNIF_HPP )
#include "randunif.hpp"
#endif


class RandNormal : public Rand
   {

   private:

      RandUniform rand1;              // independent U( 0, 1 ) streams
      RandUniform rand2;

      double meanNorm;
      double stdNorm;
      double varNorm;

   public:

      //-- constructor --//
      RandNormal( double _meanNorm, double _stdNorm, long _seed,
            char *_name = "N-N");

      //-- destructor --//
      virtual ~RandNormal( void ) { }

      //-- returns a normal random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:34
//---------------------------------------------------------------------
//  PROGRAM: randnorm.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDNORM_HPP )
#include "randnorm.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}



//-- constructors --//

RandNormal::RandNormal( double _meanNorm, double _stdNorm,
      long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     rand1( (UINT) 1, _seed ), // use stream 1, these should be independent!
     rand2( (UINT) 2, _seed ), // use stream 2, so use different streams
     meanNorm( _meanNorm ),
     stdNorm( fabs( _stdNorm ) ),
     varNorm( stdNorm * stdNorm )
   {
   SetDistName( "Normal" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", meanNorm );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", stdNorm );
   SetParameterB_Name( temp );
   }


// polar method by Marsaglia & Bray (1964)
double RandNormal::RandValue( void )
   {
   AddOneObservation();

   double firstValue;
   static double secondValue;    // uses both random numbers produced
   static int secondValueFlag;   // by the polar method

   double W;         // algorithm variables
   double V1;
   double V2;

   if ( secondValueFlag == FALSE )  // No second value available
      {
      secondValueFlag = TRUE;

      do    // these are not automatically included in observations
         {  // use 2 different streams, pick two uniform numbers fron -1 to 1.
         V1 = 2 * rand1.RandValue() - 1.;
         V2 = 2 * rand2.RandValue() - 1.;
         W = V1 * V1 + V2 * V2;
         }
      while ( W >= 1. );            // Make sure they're in the unit circle.

      double Y = sqrt( -2. * ::log( W ) / W ); // Box-Muller transformation.

      firstValue =  V1 * Y;
      secondValue = V2 * Y;               // Save one number for next time.

      return( stdNorm * firstValue + meanNorm );
      }

   else
      {                                  // Extra value available.
      secondValueFlag = FALSE;           // Reset the secondValueFlag.

      return( stdNorm * secondValue + meanNorm );
      }

   }
```

```
Print date: 8/17/1992     Print time:  8:26
//-----------------------------------------------------------------
// PROGRAM: randpois.hpp
//
// PURPOSE: Generates poisson distributed random numbers
//-----------------------------------------------------------------

/*
-------------------------------------------------------------------
Note: Formula derived from Law & Kelton.
      1. Let a = e ^-y, b = 1, and i = 0;
      2. Generate U(i+1) ~ U( 0, 1 ) and replace b by b*U(i+1).
         If b < a, return X = i. Otherwise, go to step 3.
      3. Replace i by i+1 and go back to step 2. [ Law & Kelton, 1991 ]

Parameters: The mean lamda is a positive real number.

Range: [ 0, 1, 2, ..., ]

Mean: lamda
Var:  lamda

   "The Poisson distribution is a discrete distribution that is often
used to model the number of random events occurring in an interval
of time. If the time between events is exponentially distributed,
then the number of events that occur in a fixed time interval has a
Poisson distribution. The Poisson distribution is also used to model
random variations in batch size." [ Pegden et al, 1990 ]

Reference: Introduction to Simulation using SIMAN by C. Dennis Pegden,
           Robert E. Shannon, and Randall P. Sadowski, 1990., pg., 565.

           Simulation Modeling & Analysis by Averill M. Law and W. David
           Kelton, 1991, pg., 503.
-------------------------------------------------------------------
*/

#if !defined _RANDPOIS_HPP
#define      _RANDPOIS_HPP 1

//--------------------------- Includes --------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandPoisson : public Rand
   {

   private:

      double mean;

      double a;                 // algorithm coefficients, See Law & Kelton

   public:

      //-- constructors --//
      RandPoisson( double _mean, char *_name = "-" );
      RandPoisson( double _mean, UINT _stream, long _seed, char *_name = "-" )

      //-- destructor --//
      virtual ~RandPoisson( void ) { }

      //-- returns a poisson random number --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:37
//---------------------------------------------------------------
//  PROGRAM: randpois.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDPOIS_HPP )
#include "randpois.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandPoisson::RandPoisson( double _mean, char *_name )
   : Rand( 1, _name ),
     mean( fabs( _mean ) ),
     a( exp( - mean ) )

   {
   SetDistName( "Poisson" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", mean );
   SetParameterA_Name( temp );
   }

RandPoisson::RandPoisson( double _mean, UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     mean( fabs( _mean ) ),
     a( exp( - mean ) )

   {
   SetDistName( "Poisson" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", mean );
   SetParameterA_Name( temp );
   }


double RandPoisson::RandValue( void )
   {
   AddOneObservation();

   double b = 1;
   long count = 0;

   //-- count observations once --//

   b = b * GetUnif01();

   if ( b < a )
      return ( (double) count );

   do
      {
      count++;
      b = b * GetUnif01();
      } while( b >= a );

   return ( (double) count );
   }
```

```
Print date: 8/17/1992     Print time:  8:29
//-------------------------------------------------------------------
// PROGRAM: randtri.hpp
//
// PURPOSE: Generates triangular distributed random numbers
//
// NOTE:    Composition method
//          tMin <= tMode < tMax     or     tMin < tMode <= tMax
//          The constructor cannot check that this is valid !!
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
   Note: Inverse transform method:
   If X ~ triangular( 0, 1, ( max - min ) / ( mode - min ) )
   then X' = min + ( mode - min ) * X   is equivalent to
   triang( min, mode, max ) [ Law & Kelton, 1982 ]

   Formula obtained from Hoover & Perry, see reference. Composition
   method.

   Note: There may be a problem with independence in this approach!
         ie., I'm not sure if my samples are independent.


Parameters: The min, mode, and max values for the distribution
            specified as real numbers with a min < mode < max

Range:      [ min, max ]

Mean: ( min + mode + max ) / 3

Var: ( min^2 + mode^2 + max^2 - mode * min - min * max - mode * max ) / 18

   "The triangular distribution is commonly used in situations in which
the exact form of the distribution is not known, but estimates for the
minimum, maximum, and most likely values are available. The triangular
distribution is easier to use and explain than other distributions that
may be used in this situation ( e.g., the beta distribution )".
[ Pegden et al, 1990 ]

Reference: Simulation Modeling and Analysis by Averill M. Law and W. David
           Kelton, 1982, pg., 261

           Introduction to Simulation Using SIMAN by C. Dennis Pegden,
           Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 566.

           Simulation by Stewart V. Hoover and Ronald F. Perry, 1989,
           pg., 271-272.
---------------------------------------------------------------------
*/

#if !defined _RANDTRI_HPP
#define      _RANDTRI_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif
```

```
class RandTriangular : public Rand
   {

   private:

      //-- parameters --//
      double tMin;
      double tMode;
      double tMax;

      double S;               // algorithm coefficients; see Simulation by
      double T;               // [ Hoover and Perry, 1989 ]

   public:

      //-- constructors --//
      RandTriangular( double _tMin, double _tMode, double _tMax,
            char *_name = "-" );
      RandTriangular( double _tMin, double _tMode, double _tMax,
         UINT _stream, long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandTriangular( void ) { }

      //-- returns a triangular random number --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:38
//---------------------------------------------------------------
//  PROGRAM: randtri.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDTRI_HPP )
#include "randtri.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandTriangular::RandTriangular( double _tMin, double _tMode,
      double _tMax, char *_name )
   : Rand( 1, _name ),     // stream = 1
     tMin( _tMin ),
     tMode( _tMode ),
     tMax( _tMax ),
     S( tMode - tMin ),
     T( tMax - tMin )
   {
   SetDistName( "Triang" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", tMin );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", tMode );
   SetParameterB_Name( temp );
   sprintf( temp, "%g", tMax );
   SetParameterC_Name( temp );
   }


RandTriangular::RandTriangular( double _tMin, double _tMode, double _tMax,
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     tMin( _tMin ),
     tMode( _tMode ),
     tMax( _tMax ),
     S( tMode - tMin ),
     T( tMax - tMin )
   {
   SetDistName( "Triang" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", tMin );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", tMode );
   SetParameterB_Name( temp );
   sprintf( temp, "%g", tMax );
   SetParameterC_Name( temp );
   }


double RandTriangular::RandValue( void )
   {
   AddOneObservation();

   // composition method at the moment, from Perry and Hoover.

   //-- generate a uniform 0-1 random variable
   double U01A = GetUnif01();          // uses stream 1
   double U01B = GetUnif01();          // independent ???
   double X = 0;

   if ( U01A <= S/T )
      X = S * sqrt( U01B ) + tMin;
   else
      X = T - ( T - S ) * sqrt( U01B ) + tMin;

   return( X );
   }
```

```
Print date: 8/17/1992     Print time:  8:25
//---------------------------------------------------------------------
// PROGRAM: randunif.hpp
//
// PURPOSE: Generates uniform ( 0, 1 ) random numbers
//          or uniform ( lowerBound, upperBound ) random numbers
//          where lowerBound < upperBound
//---------------------------------------------------------------------

/*
-----------------------------------------------------------------------
   Note: Inverse transform method for U( lowerBound, upperBound )
   numbers is X = lowerBound + ( upperBound - lowerBound ) * U( 0,1 )
   [ Bulgren, 1982 ]

Parameters: The minimum and maximum value for the distribution specified
            as real numbers.

Range:      [ lowerBound, upperBound ]

Mean:       ( lowerBound + upperBound ) / 2

Variance:   ( ( upperBound - lowerBound ) ^ 2 ) / 12

   "The uniform distribution is used when all values over a finite range
are considered to be equally likely. It is sometimes used when no
information other than the range is available. The uniform distribution
has a larger variance than other distributions that are used when information
is lacking( e.g., the triangular distribution ). Because of its large
variance, the uniform distribution generally produces "worst case" results."
[ Pegden et al, 1990 ]


References: Discrete System Simulation by William G. Bulgren, 1982, pg. 167.

            Introduction to Simulation using SIMAN by C. Dennis Pegden,
            Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 567.
-----------------------------------------------------------------------
*/

#if !defined _RANDUNIF_HPP
#define      _RANDUNIF_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandUniform : public Rand
   {

   private:

      //-- scale the U( 0,1 ) to a new range --//
      double lowerBound;
      double upperBound;

   public:

      //-- constructors --//
      RandUniform( char *_name = "-" );
      RandUniform( UINT _stream, long _seed, char *_name = "-" );
      RandUniform( double _lowerBound, double _upperBound, long _seed,
            char *_name = "-" );
      RandUniform( double _lowerBound, double _upperBound, \
            UINT _stream, long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~RandUniform( void ) { }

      //-- return a uniform number --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:34
//---------------------------------------------------------------------
//  PROGRAM: randunif.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RANDUNIF_HPP )
#include "randunif.hpp"
#endif

extern "C"
{
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandUniform::RandUniform( char *_name )
   : Rand( 1, _name ),
     lowerBound( 0 ),
     upperBound( 1 )
   {
   SetDistName( "Uniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", upperBound );
   SetParameterB_Name( temp );
   }


RandUniform::RandUniform( UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     lowerBound( 0 ),
     upperBound( 1 )
   {
   SetDistName( "Uniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", upperBound );
   SetParameterB_Name( temp );
   }

RandUniform::RandUniform( double _lowerBound, double _upperBound,
      long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound )
   {
   SetDistName( "Uniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", upperBound );
   SetParameterB_Name( temp );
   }

RandUniform::RandUniform( double _lowerBound, double _upperBound, \
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     lowerBound( _lowerBound ),
     upperBound( _upperBound )
   {
   SetDistName( "Uniform" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", lowerBound );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", upperBound );
   SetParameterB_Name( temp );
   }


double RandUniform::RandValue( void )
   {
   AddOneObservation();

   return ( lowerBound + ( upperBound - lowerBound ) * GetUnif01() );
   }
```

```
Print date: 8/17/1992     Print time:  8:26
//-------------------------------------------------------------------
// PROGRAM: randweib.hpp
//
// PURPOSE: Generates weibull distributed random numbers
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
Note: Inverse transform method where
      X = scale * ( - ln( U( 0,1 ) ) ^ ( 1 / shape ) )
      [ Bulgren, 1982 ]

Parameters: shape parameter (alpha) and scale parameter (beta) are
            non-negative reals, ( 1 / scale ) is similar to that of
            the mean in the exponential dist.

Range:      [ 0, +infinity ]

Mean: see [ Pegden et al, 1990 ]
Var:  see [ Pegden et al, 1990 ]

   "The Weibull distribution is widely used in reliability models to
represent the lifetime of a device. If a system consists of a large number
of parts that fail independently, and if the system fails when any single
part fails, then the time between failures can be approximated by the
Weibull distribution. This distribution is also used to represent non-
negative task times that are skewed to the left". [ Pegden et al, 1990 ]


References: Discrete System Simulation by William G. Bulgren 1982, pg. 169.

            Introduction to Simulation Using SIMAN by C. Dennis Pegden,
            Robert E. Shannon, and Randall P. Sadowski, 1990, pg., 568.
---------------------------------------------------------------------
*/

#if !defined _RANDWEIB_HPP
#define      _RANDWEIB_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


class RandWeibull : public Rand
   {

   private:

      double shape;
      double scale;
      double oneOverShape;

   public:

      //-- constructors --//
      RandWeibull( double _shape, double _scale, char *_name = "-" );
      RandWeibull( double _shape, double _scale, UINT _stream, long _seed,
            char *_name = "-" );

      //-- destructor --//
      virtual ~RandWeibull( void ) { }

      //-- return a weibull random number --//
      virtual double RandValue( void );

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:33
//------------------------------------------------------------------
//  PROGRAM: randweib.cpp
//------------------------------------------------------------------

#pragma hdrstop

//--------------------------- Includes ---------------------------//

#if !defined ( _RANDWEIB_HPP )
#include "randweib.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructors --//

RandWeibull::RandWeibull( double _shape, double _scale, char *_name )
   : Rand( 1, _name ),
     shape( fabs( _shape ) ),
     scale( fabs( _scale ) ),
     oneOverShape( 1 / shape )
   {
   SetDistName( "Weibull" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );
   }


RandWeibull::RandWeibull( double _shape, double _scale,
      UINT _stream, long _seed, char *_name )
   : Rand( _stream, _seed, _name ),
     shape( fabs( _shape ) ),
     scale( fabs( _scale ) ),
     oneOverShape( 1 / shape )
   {
   SetDistName( "Weibull" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%g", shape );
   SetParameterA_Name( temp );
   sprintf( temp, "%g", scale );
   SetParameterB_Name( temp );
   }


double RandWeibull::RandValue( void )
   {

AddOneObservation();

return ( scale * pow( -log( GetUnif01() ), oneOverShape ) );
}
```

```
Print date: 8/17/1992     Print time:  8:26
//---------------------------------------------------------------
// PROGRAM: rand_f.hpp
//
// PURPOSE: Generates "F" distributed random numbers
//          with ( k1, k2 ) degrees of freedom
//---------------------------------------------------------------

/*
-----------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   If Z1 ~ Chi^2 with k1 df, and Z2 ~ Chi^2 with k2 df, and Z1 and Z2
   are independent, X = ( Z1 / k1 ) / ( Z2 / k2 ) is said to have an
   F distribution with ( k1, k2 ) degrees of freedom. We can write
   X ~ F(k1,k2).


Parameters: k1 and k2 are positive integers that are referred to as the
            degrees of freedom.

Range:      [ 0 , +infinity ]

Mean:       k2 / ( k2 - 2 ),  where k2 > 2

Var:        2 * k2 * k2 * ( k1 + k2 - 2 ) / k1 * ( k2 - 2 )^2 * ( k2 - 4 )

   The F distribution is sometimes referred to as the variance ratio.


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 485, 682.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 173.

            Statisticsl Distributions bye Hastings and Peacock, 1975,
            pgs., 64-67.
-----------------------------------------------------------------
*/

#if !defined _RAND_F_HPP
#define      _RAND_F_HPP 1

//---------------------------- Includes -------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


//----------------------- Forward reference --------------------//

class RandChiSquare;


class Rand_F : public Rand
   {
   private:

      RandChiSquare  *pRNG1;        // with k1 df
      RandChiSquare  *pRNG2;        // with k2 df

      UINT k1;                      // k1 degrees of freedom
      UINT k2;                      // k2 degrees of freedom

   public:

      //-- constructor --//
      Rand_F( UINT _k1, UINT _k2, UINT _stream, long _seed, char *_name = "-"

      //-- destructor --//
      virtual ~Rand_F( void );

      //-- returns a F distributed random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:32
//---------------------------------------------------------------
//  PROGRAM: rand_f.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RAND_F_HPP )
#include "rand_f.hpp"
#endif

#if !defined ( _RANDCHI_HPP )
#include "randchi.hpp"
#endif

extern "C"
{
#include <string.h>
#include <stdio.h>
}


//-- constructor --//
Rand_F::Rand_F( UINT _k1, UINT _k2, UINT _stream, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     pRNG1( NULL ),
     pRNG2( NULL ),
     k1( _k1 ),
     k2( _k2 )
   {
   SetDistName( "F-dist" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", k1 );
   SetParameterA_Name( temp );
   sprintf( temp, "%u", k2 );
   SetParameterB_Name( temp );

   //-- run the constructors for the chi-square distribution --//
   pRNG1 = new RandChiSquare( k1, _stream, _seed );
   pRNG2 = new RandChiSquare( k2, _stream + 1, _seed );
   }


//-- destructor --//
Rand_F::~Rand_F( void )
   {
   if ( pRNG1 )
      delete pRNG1;

   if ( pRNG2 )
      delete pRNG2;
   }


double Rand_F::RandValue( void )
   {

AddOneObservation();

double Numerator   = pRNG1->RandValue() / (double) k1;
double Denominator = pRNG2->RandValue() / (double) k2;

return( Numerator / Denominator );
}
```

```
Print date: 8/17/1992     Print time:  8:27
//---------------------------------------------------------------------
// PROGRAM: rand_t.hpp
//
// PURPOSE: Generates Students "t" distributed random numbers
//          with k degrees of freedom
//
// NOTE:    We make use of the gamma to get Chi-squared variables
//---------------------------------------------------------------------

/*
-----------------------------------------------------------------------
   Note: Formula obtained from Law & Kelton, see reference.
   If X ~ N( 0, 1 ) and Y has a Chi^2 with k degrees of freedom, then
   X' ~ X / sqrt( Y / k ) has a t distribution with k degrees of
   freedom.


Parameters: k degreesOfFreedom

Range:      [ -infinity, +infinity ]

Mean:       0

Var:        k / ( k - 2 ),  where k > 2


References: Simulation Modeling & Analysis by Law and Kelton, 1991,
            pgs., 288, 291, 336, 485, 560.

            Discrete System Simulation by William G. Bulgren, 1982,
            pg., 173.

            Statisticsl Distributions bye Hastings and Peacock, 1975,
            pgs., 120-123.
-----------------------------------------------------------------------
*/

#if !defined _RAND_T_HPP
#define      _RAND_T_HPP 1

//---------------------------- Includes --------------------------//

#if !defined ( _RAND_HPP )
#include "rand.hpp"
#endif


//----------------------- Forward references --------------------//

class RandGamma;
class RandNormal;


class Rand_T : public Rand
   {

   private:

      RandGamma      *pRNGGamma;     // with k df
      RandNormal     *pRNGNorm;      // Normal( 0, 1 )

      UINT degreesOfFreedom;         // k df

   public:

      //-- constructor --//
      Rand_T( UINT _degreesOfFreedom, long _seed, char *_name = "-" );

      //-- destructor --//
      virtual ~Rand_T( void );

      //-- returns a log normal random variable --//
      virtual double RandValue( void );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:32
//---------------------------------------------------------------
//  PROGRAM: rand_t.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _RAND_T_HPP )
#include "rand_t.hpp"
#endif

#if !defined ( _RANDGAMA_HPP )
#include "randgama.hpp"
#endif

#if !defined ( _RANDNORM_HPP )
#include "randnorm.hpp"
#endif

extern "C"
{
#include <math.h>
#include <string.h>
#include <stdio.h>
}


//-- constructor --//
Rand_T::Rand_T( UINT _degreesOfFreedom, long _seed, char *_name )
   : Rand( 1, _seed, _name ),
     pRNGGamma( NULL ),
     pRNGNorm( NULL ),
     degreesOfFreedom( _degreesOfFreedom )
   {
   SetDistName( "T-dist" );

   //-- set the coefficients --//
   char temp[ 40 ];
   sprintf( temp, "%u", degreesOfFreedom );
   SetParameterA_Name( temp );

   //-- run the constructor for the gamma distribution --//
   pRNGGamma = new RandGamma( (double) degreesOfFreedom / 2.0, 2.0, _seed );

   //-- run the constructor for the normal distribution --//
   pRNGNorm = new RandNormal( 0, 1, _seed );
   }


//-- destructor --//
Rand_T::~Rand_T( void )
   {
   if ( pRNGGamma )
      delete pRNGGamma;

   if ( pRNGNorm )
      delete pRNGNorm;
   }


double Rand_T::RandValue( void )
   {
   AddOneObservation();

   double Y = pRNGNorm->RandValue();
   double Z = pRNGGamma->RandValue();

   double X = Y / ( sqrt( Z / (double) degreesOfFreedom ) );

   return( X );
   }
```

Appendix I: Statistical collection classes

```
Print date: 8/17/1992     Print time:  8:20
//---------------------------------------------------------------
// PROGRAM: stats.h
//
// PURPOSE: Statistics class definitions
//---------------------------------------------------------------

#if !defined _STATS_H
#define      _STATS_H  1

typedef unsigned int  UINT;

#endif
```

```
Print date: 8/17/1992     Print time:  8:21
//-------------------------------------------------------------------
// PROGRAM: stats.hpp
//
// PURPOSE: Statistics class
//-------------------------------------------------------------------

/*
---------------------------------------------------------------------
Note: When conducting performance analysis in simulation, statistics
      are either time based, continuous based, observational based, or
      some counting statistic. Some different types follow.

Observational:    Throughput
                  Makespan ( time in system )
                  Time in a queue for all
                  Time in a queue for those who wait
                  Time per transaction
                  Busy time per server
                  Idle time per server
                  Blockage time

Time based:       Utilization
                  Bottleneck analysis
                    1. Queue length
                    2. Proportion busy
                    3. Proportion blocked

Continuous based: Any statistics on continuously changing variables

Counters:         Number of entries

   The children of the Stats abstract class are:
      class StatsObservational
      class StatsTime
      class StatsContinuous ( child of StatsTime )

      class Counter is a seperate class that collect count statistics
---------------------------------------------------------------------
*/

#if !defined _STATS_HPP
#define      _STATS_HPP  1

#if !defined _STATS_H
#include "stats.h"
#endif

extern "C"
{
#include <string.h>
}


class Stats
   {

   protected:

      UINT id;                  // statistic id
      char *name;               // statistic name
      long observations;        // number of times stats recorded

      double value;             // statistic value recorded

      double minimum;
      double maximum;

      double Sx;                // Sum of X's or sum of values
      double SSx;               // Sum square of X's or sum square of values

      void SetRange( void );    // set the minimum and maximum observation

      virtual void SetCurrentTime( double _currentTime ) { return; }
      virtual void SetValue( double _value ) { value = _value; }
      virtual void SetSx( void ) = 0;
      virtual void SetSSx( void ) = 0;
      virtual void UpdateLastTime( double _currentTime ) { return; }

   public:

      //-- constructor --//
      Stats( UINT _id, char *_name );

      //-- destructor --//
      virtual ~Stats( void );

      //-- record observation --//
      void RecordValue( double _value );
      void RecordValue( double _value, double _currentTime );

      //-- pure virtual --//
      virtual double GetMean( void ) = 0;
      virtual double GetVariance( void ) = 0;
      virtual double GetVariation( void ) = 0;

      //-- --//
      double GetMinimum( void );
      double GetMaximum( void );
      long   GetObservations( void ) { return observations; }

      double GetSx( void ) { return Sx; }
      double GetSSx( void ){ return SSx; }

      UINT GetID   ( void ) { return id; }
      const char* const GetName( void ) { return (const char* const) name; }
      void SetName( char *_name );

      double GetValue( void ) { return value; }

      virtual void ReInitializeStats( double _currentTime );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:23
//---------------------------------------------------------------
// PROGRAM: stats.cpp
//---------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _STATS_HPP )
#include "stats.hpp"
#endif

extern "C"
{
#include <math.h>
}


//-- constructor --//
Stats::Stats( UINT _id, char *_name )
    : id( _id ),
      name( NULL ),
      observations( 0 ),
      value( 0 ),
      minimum( 1.7e30 ),
      maximum( -1.7e30 ),
      Sx( 0 ),
      SSx( 0 )
   {
   SetName( _name );
   }


Stats::~Stats( void )
   {
   if ( name )
      delete name;
   }


void Stats::SetName( char  *_name )
   {
   if ( name )
      delete name;

   if ( _name )
      {
      name = new char[ strlen( _name ) + 1 ];
      strcpy( name, _name );
      }
   else
      name = NULL;
   }


//-- a value for the statistic --//
void Stats::RecordValue( double _value )
   {
   observations++;

   value = _value;
   SetSx();
   SetSSx();
   SetRange();

   return;
   }

//-- a value for the statistic --//
void Stats::RecordValue( double _value, double _currentTime )
   {
   observations++;
   SetValue( _value );
   SetCurrentTime( _currentTime );
   SetSx();
   SetSSx();
   SetRange();
   UpdateLastTime( _currentTime );

   return;
   }


double Stats::GetMinimum( void )
   {
   if ( minimum >= 1.7e30 )
      return 0;
   else
      return minimum;
   }


double Stats::GetMaximum( void )
   {
   if ( maximum <= -1.7e30 )
      return 0;
   else
      return maximum;
   }


void Stats::ReInitializeStats( double _currentTime )
   {
   minimum      = 1.7e30;
   maximum      = - 1.7e30;
   Sx           = 0;
   SSx          = 0;
   observations = 0;

   return;
   }

// set the minimum and maximum observation
void Stats::SetRange( void )
   {
   if( value <= minimum )
      minimum = value;

   if( value >= maximum )
      maximum = value;
```

```
return;
}
```

```
Print date: 8/17/1992     Print time:  8:21
//-------------------------------------------------------------------
// PROGRAM: stat_tim.hpp
//
// PURPOSE: Time based statistics class
// ------------------------------------------------------------------

/*
--------------------------------------------------------------------
   Time based statistics are statistics whose values are associated
with some proportion of time. Queue length is a time based statistic
because we cannot speak of the length of the queue without associating
some time frame with it, ie., the queue was size 10 for the first 5
minutes and then size 8 for the next 4 minutes.
--------------------------------------------------------------------
*/

#if !defined _STAT_TIM_HPP
#define      _STAT_TIM_HPP  1

//-------------------------- Includes --------------------------//

#if !defined ( _STATS_HPP )
#include "stats.hpp"
#endif


class StatsTime : public Stats
   {

   protected:

      double startTime;       // start time of statistics gathering
      double currentTime;     // current time of this observation
      double lastTime;        // last time observation was taken

      virtual void SetCurrentTime( double _currentTime )
         { currentTime = _currentTime; }

      virtual void SetValue( double _value ) { value = _value; }

      virtual void UpdateLastTime( double _currentTime )
         { lastTime = _currentTime; }

      virtual void SetSx( void );
      virtual void SetSSx( void );

   public:

      //-- constructor --//
      StatsTime( UINT _id, char *_name, double _currentTime );

      //-- destructor --//
      virtual ~StatsTime( void ) { };

      double TotalTime( void );

      virtual double GetMean( void );
      virtual double GetVariance( void );

      virtual double GetVariation( void );

      virtual void ReInitializeStats( double _currentTime );

      };

#endif
```

```
Print date: 8/17/1992     Print time:  8:22
//-------------------------------------------------------------------
// PROGRAM: stat_tim.cpp
//-------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _STAT_TIM_HPP )
#include "stat_tim.hpp"
#endif

extern "C"
{
#include <math.h>
}


//-- constructor --//
StatsTime::StatsTime( UINT _id, char *_name, double _startTime )
   : Stats( _id, _name ),
     startTime( _startTime ),
     currentTime( _startTime ),
     lastTime( _startTime )
   { }


double StatsTime::TotalTime( void )
   { return ( currentTime - startTime ); }


void StatsTime::SetSx( void )
   {
   Sx = Sx + value * ( currentTime - lastTime );

   return;
   }


void StatsTime::SetSSx( void )
   {
   SSx = SSx + ( value * value ) * ( currentTime - lastTime );

   return;
   }


double StatsTime::GetMean( void )
   {
   if ( TotalTime() == 0 )
      return 0;
   else
      return( Sx / TotalTime() );
   }


double StatsTime::GetVariance( void )
   {
   if ( TotalTime() == 0 )
      return 0;
   else
      {
      double mean = GetMean();

      return( ( SSx / TotalTime() ) -  ( mean * mean ) );
      }
   }


double StatsTime::GetVariation( void )
   {
   double mean = GetMean();
   double variance = GetVariance();

   if ( variance < 0.0 || mean == 0 )  // run time error check
      return -1;

   double standardDeviation = sqrt( variance );

   return ( standardDeviation / mean );
   }


void StatsTime::ReInitializeStats( double _currentTime )
   {
   Stats::ReInitializeStats( _currentTime );

   startTime   = _currentTime;
   currentTime = _currentTime;
   lastTime    = _currentTime;

   return;
   }
```

```
Print date: 8/17/1992     Print time:  8:21
//-------------------------------------------------------------------
// PROGRAM: stat_con.hpp
//
// PURPOSE: Continuous based statistics class
// ------------------------------------------------------------------

/*
-------------------------------------------------------------------
   Continuous time statistics resemble discrete time statistics in form
and function except in their representation of the area under a curve.
Note that in the discrete case, the area under some level for a statistic
is a rectangle and easily calculated. In the case of a continuously
changing variable, the area under the curve cannot be approximated by a
rectangle. Sums and Sums of squares can be collected on continuously
changing variables easily if we use the trapazoidal approximation rule.
This rule, allow not perfect, is considerable better than using rectangles
which would over or under estimate values when the curve was steep. The
area of a trapazoid is easily calculated.
-------------------------------------------------------------------
*/

#if !defined _STAT_CON_HPP
#define      _STAT_CON_HPP  1

//-------------------------- Includes --------------------------//

#if !defined ( _STAT_TIM_HPP )
#include "stat_tim.hpp"
#endif


class StatsContin : public StatsTime
   {

   protected:

      double lastValue;       // used to compute the area of a trapazoid

      virtual void SetValue( double newValue );

      virtual void SetSx( void );
      virtual void SetSSx( void );

   public:

      //-- constructor --//
      StatsContin( UINT _id, char *_name, double _currentTime,
            double _initialValue );

      //-- destructor --//
      virtual ~StatsContin( void ) { }

      virtual void ReInitializeStats( double _currentTime );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:23
//-------------------------------------------------------------------
// PROGRAM: stat_con.cpp
//-------------------------------------------------------------------

#pragma hdrstop

//--------------------------- Includes ---------------------------//

#if !defined ( _STAT_CON_HPP )
#include "stat_con.hpp"
#endif

extern "C"
{
#include <math.h>
}


//-- constructor --//
StatsContin::StatsContin( UINT _id, char *_name,
     double _startTime, double _initialValue )
   : StatsTime( _id, _name, _startTime ),
     lastValue( 0 )
   {
   value = _initialValue;      // initialize grandparent data
   SetRange();                 // set min and max to current value
   }


void StatsContin::SetValue( double newValue )
   {
   lastValue = value;
   value = newValue;

   return;
   }

void StatsContin::SetSx( void )
   {
   //-- sum area of trapazoids --//
   Sx = Sx + ( value + lastValue ) / 2 * ( currentTime - lastTime );

   return;
   }


void StatsContin::SetSSx( void )
   {
   SSx = SSx + (( value + lastValue ) / 2) * (( value + lastValue ) / 2) *
      ( currentTime - lastTime );

   return;
   }


void StatsContin::ReInitializeStats( double _currentTime )
   {
   StatsTime::ReInitializeStats( _currentTime );

   return;
   }
```

```
Print date: 8/17/1992     Print time:  8:21
//-------------------------------------------------------------------
// PROGRAM: stat_obs.hpp
//
// PURPOSE: Observational statistics class
// ------------------------------------------------------------------

/*
--------------------------------------------------------------------
   Observational based statistics are summary statistics collected on
some variable which is not time based. This may seem confusing when a
statistic like 'Time in the system' is collected but note that the value
'time in the system' is not multiplied by any proportion of time to
calculate a value for the statistic. If one keeps in mind that all time
based statistics are multiplied by the proportion of time they were at
this value, then the use of observational based statistics should not
become confused with time based statistics. Observational based
statistics are observations at a point in time. The proportion of time
the statistic is at this value is not important.
--------------------------------------------------------------------
*/


#if !defined _STAT_OBS_HPP
#define      _STAT_OBS_HPP  1

//-------------------------- Includes --------------------------//

#if !defined ( _STATS_HPP )
#include "stats.hpp"
#endif


class StatsObs : public Stats
   {

   protected:

      void SetSx( void );
      void SetSSx( void );

   public:

      //-- constructor --//
      StatsObs( UINT _id, char *_name ) : Stats( _id, _name ) { }

      //-- destructor --//
      virtual ~StatsObs( void ) { }

      virtual double GetMean( void );
      virtual double GetVariance( void );
      virtual double GetVariation( void );

      virtual void ReInitializeStats( double _currentTime );

   };

#endif
```

```
Print date: 8/17/1992     Print time:  8:23
//---------------------------------------------------------------------
// PROGRAM: stat_obs.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _STAT_OBS_HPP )
#include "stat_obs.hpp"
#endif

extern "C"
{
#include <math.h>
}

//-- sum each statistical observation --//
void StatsObs::SetSx( void )
   {
   Sx = Sx + value;

   return;
   }

//-- sum the square of each statistical observation --//
void StatsObs::SetSSx( void )
   {
   SSx = SSx + value * value;

   return;
   }

//-- return the mean of this statistic --//
double StatsObs::GetMean( void )
   {
   if ( observations == 0 )
      return 0;
   else
      return( Sx / (double) observations );
   }

//-- return the variance of this statistic --//
double StatsObs::GetVariance( void )
   {
   if ( observations == 0 || observations == 1 )
      return 0;
   else
      {
      double mean = GetMean();

      return ( ( SSx - (double) observations * mean * mean ) /
            (double) observations );
      }
   }

//-- return the variation of this statistic --//
double StatsObs::GetVariation( void )
   {
   double mean = GetMean();
   double variance = GetVariance();

   if ( variance < 0 || mean == 0 )    // run time error check
      return -1;

   double standardDeviation = sqrt( variance );

   return ( standardDeviation / mean );
   }


void StatsObs::ReInitializeStats( double _currentTime )
   {
   Stats::ReInitializeStats( _currentTime );

   return;
   }
```

```
Print date: 8/17/1992     Print time:  8:21
//-----------------------------------------------------------------
// PROGRAM: counter.hpp
//
// PURPOSE: Statistics counter class
//
// NOTE: Counters are very simple in that all they do is count the
//       values given to them. Their added functionallity allows them
//       to be incorporated into a simulation language.
//-----------------------------------------------------------------

#if !defined _COUNTER_HPP
#define      _COUNTER_HPP  1

#if !defined ( _STATS_H )
#include "stats.h"
#endif


class Counter
   {

   protected:

      UINT id;                  // statistic id
      char *name;               // statistic name
      long observations;        // number of times stats recorded
      long count;               // statistic value recorded

   public:

      //-- constructor --//
      Counter( UINT _id, char *_name );

      //-- destructor --//
      ~Counter( void );

      void RecordValue( long value );

      long GetObservations( void ) { return observations; }
      long GetCount( void )        { return count; }

      UINT GetID   ( void ) { return id; }
      const char* const GetName( void ) { return (const char* const) name; }
      void SetName( char *_name );

      void ReInitializeStats( void ) { observations = 0; count = 0; }

   };


#endif
```

```
Print date: 8/17/1992     Print time:  8:22
//---------------------------------------------------------------------
// PROGRAM: counter.cpp
//---------------------------------------------------------------------

#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _COUNTER_HPP )
#include "counter.hpp"
#endif

extern "C"
{
#include <string.h>
}


//-- constructor --//
Counter::Counter( UINT _id, char *_name )
   : id( _id ),
     name( NULL ),
     observations( 0 ),
     count( 0 )
   {
   SetName( _name );
   }


Counter::~Counter( void )
   {
   if ( name )
      delete name;
   }


void Counter::SetName( char  *_name )
   {
   if ( name )
      delete name;

   if ( _name )
      {
      name = new char[ strlen( _name ) + 1 ];
      strcpy( name, _name );
      }
   else
      name = NULL;
   }


void Counter::RecordValue( long value )
   {
   observations++;
   count += value;

   return;
   }
```

Appendix J: Replication statistics class ReplicationStats

```
Print date: 8/17/1992     Print time:  8:18
//-------------------------------------------------------------------
// PROGRAM: simgloba.hpp
//
// PURPOSE: Simulation global statistics class
//
// NOTE: These statistics are created during a multiple run
//       or replication simulation experiment.
//-------------------------------------------------------------------

#if !defined _SIMGLOBA_HPP
#define      _SIMGLOBA_HPP  1


//-------------------- Forward Declarations ----------------------//

class StatSimEnv;

#if !defined ( _ST_ARRAY_HPP )
#include "st_array.hpp"
#endif

#if !defined ( _TMATRIX_HPP )
#include "tmatrix.hpp"
#endif

#if !defined ( _STATS_HPP )
#include "stats.hpp"
#endif


class SimEnvGlobalStats
   {
   friend StatSimEnv;  // for now

   protected:

//      //-- array lengths --//  ! moved to StatSimEnv class
//      UINT repNum;          // current replication number
      StatSimEnv *pStatSimEnv;

      UINT distLength;
      UINT obsLength;
      UINT timeLength;
      UINT continLength;
      UINT countLength;

      //-- 95% t distribution look-up table --//
      static float tTable[];

      //-- row counters for arrays --//
      UINT distRow;
      UINT continRow;
      UINT timeRow;
      UINT obsRow;
      UINT countRow;
      UINT seedRow;

      //-- global statistic arrays --//

      long  *distObs;
      long  *initialSeed;     // stores initial seeds of 1st rep
      long  *lastSeed;

      long  *obsObs;
      float *obsMax;
      float *obsMin;
      TMatrix < float > *obsAve;
      float *obsMean;
      float *obsStDev;
      float *obsHalfWidth;

      long  *timeObs;
      float *timeMax;
      float *timeMin;
      TMatrix < float > *timeAve;
      float *timeMean;
      float *timeStDev;
      float *timeHalfWidth;

      long  *continObs;
      float *continMax;
      float *continMin;
      TMatrix < float > *continAve;
      float *continMean;
      float *continStDev;
      float *continHalfWidth;

      long  *countObs;

   public:

      //-- constructor --//
      SimEnvGlobalStats( StatSimEnv *_pStatSimEnv );

      //-- destructor --//
      ~SimEnvGlobalStats( void );

      float Get_T_Statistic( UINT numReps );

      float ComputeHalfWidth( float *arrayOfMeans, UINT lengthOfArray );

      float ComputeMean( float *arrayOfMeans, UINT lengthOfArray );

      float ComputeStDev( float *arrayOfMeans, UINT lengthOfArray );

      //-- set-up the global storage arrays --//
      void Setup( UINT numReps, UINT distLength, UINT obsLength,
           UINT timeLength, UINT continLength, UINT countLength );

      //-- prepare output arrays --//
      void SetUpGlobalStatsOutput( UINT _numReps );

      //-- set the global statistics --//
      void SetGlobalDistStats( DistArray *pDistArray );
      void SetInitialSeeds( DistArray *pDistArray );

      void SetContinMax( float maxValue );
      void SetContinMin( float minValue );
      void SetGlobalContinStats( StatsContinArray *pContinArray );
```

```
        void SetTimeMax( float maxValue );
        void SetTimeMin( float minValue );
        void SetGlobalTimeStats( StatsTimeArray *pTimeArray );

        void SetObsMax( float maxValue );
        void SetObsMin( float minValue );
        void SetGlobalObsStats( StatsObsArray *pObsArray );

        void SetGlobalCountStats( CounterArray *pCountArray );
        void SetGlobalDistSeeds( DistArray *pDistArray );

        void InitializeDistRow( void ) { distRow = 0; }
        void InitializeContinRow( void ) { continRow = 0; }
        void InitializeTimeRow( void ) { timeRow = 0; }
        void InitializeObsRow( void ) { obsRow = 0; }
        void InitializeCountRow( void ) { countRow = 0; }


        //-- get pointers to the arrays that store the global statistics --//

        long  GetDistObs( UINT _rowCount ) { return distObs[ _rowCount ]; }
        long  GetLastSeed( UINT _rowCount ) { return lastSeed[ _rowCount ]; }
        long  GetInitialSeed( UINT _rowCount ) { return initialSeed[ _rowCount ];

        long  GetObsObs( UINT _rowCount ) { return obsObs[ _rowCount ]; }
        float GetObsMax( UINT _rowCount ) { return obsMax[ _rowCount ]; }
        float GetObsMin( UINT _rowCount ) { return obsMin[ _rowCount ]; }
        float GetObsMean( UINT _rowCount ) { return obsMean[ _rowCount ]; }
        float GetObsStDev( UINT _rowCount ) { return obsStDev[ _rowCount ]; }
        float GetObsHalfWidth( UINT _rowCount ) { return obsHalfWidth[ _rowCount ]

        long  GetTimeObs( UINT _rowCount ) { return timeObs[ _rowCount ]; }
        float GetTimeMax( UINT _rowCount ) { return timeMax[ _rowCount ]; }
        float GetTimeMin( UINT _rowCount ) { return timeMin[ _rowCount ]; }
        float GetTimeMean( UINT _rowCount ) { return timeMean[ _rowCount ]; }
        float GetTimeStDev( UINT _rowCount ) { return timeStDev[ _rowCount ]; }
        float GetTimeHalfWidth( UINT _rowCount ) { return timeHalfWidth[ _rowCount

        long  GetContinObs( UINT _rowCount ) { return continObs[ _rowCount ]; }
        float GetContinMax( UINT _rowCount ) { return continMax[ _rowCount ]; }
        float GetContinMin( UINT _rowCount ) { return continMin[ _rowCount ]; }
        float GetContinMean( UINT _rowCount ) { return continMean[ _rowCount ]; }
        float GetContinStDev( UINT _rowCount ) { return continStDev[ _rowCount ];
        float GetContinHalfWidth( UINT _rowCount ) { return continHalfWidth[ _rowC

        long  GetCountObs( UINT _rowCount ) { return countObs[ _rowCount ]; }

        UINT GetDistLength( void ) { return distLength; }
        UINT GetObsLength( void ) { return obsLength; }
        UINT GetTimeLength( void ) { return timeLength; }
        UINT GetContinLength( void ) { return continLength; }
        UINT GetCountLength( void ) { return countLength; }

    };

#endif
```

```
Print date: 8/17/1992     Print time:  8:18
//-------------------------------------------------------------------
//  PROGRAM: simgloba.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop


//------------------------ Includes ----------------------------//

#if !defined ( _SIMGLOBA_HPP )
#include <simgloba.hpp>
#endif

#if !defined ( _RANDOM_HPP )
#include <random.hpp>
#endif

#if !defined ( __MATH_H )
extern "C"
{
#include <math.h>
}
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif


   //-- 95% ( .975 ) t distribution look-up table --//
float SimEnvGlobalStats::tTable[]=
   { 12.706,  4.303,  3.182,  2.776,  2.571,  2.447,  2.365,  2.306,
     2.262,  2.228,  2.201,  2.179,  2.160,  2.145,  2.131,  2.120,
     2.110,  2.101,  2.093,  2.086,  2.080,  2.074,  2.069,  2.064,
     2.060,  2.056,  2.052,  2.048,  2.045,  2.042,  2.040,  2.037,
     2.035,  2.033,  2.031,  2.029,  2.027,  2.025,  2.023,  2.021,
     2.020,  2.019,  2.017,  2.016,  2.015,  2.013,  2.012,  2.011,
     2.010,  2.009,  2.008,  2.007,  2.006,  2.005,  2.005,  2.004,
     2.003,  2.002,  2.001,  2.001,  2.000,  1.999,  1.999,  1.998,
     1.998,  1.997,  1.996,  1.996,  1.995,  1.995,  1.994,  1.994,
     1.993,  1.993,  1.992,  1.992,  1.992,  1.991,  1.991,  1.991,
     1.990,  1.990,  1.989,  1.989,  1.989,  1.988,  1.988,  1.988,
     1.987,  1.987,  1.987,  1.987,  1.986,  1.986,  1.986,  1.985,
     1.985,  1.985,  1.985,  1.984,  1.960 );

   //-- note that 100 df = 1.984, 200 df = 1.972, 300 df = 1.968
   //            400 df = 1.966, 500 df = 1.965, 1000 df = 1.962
   //   so the last entry in the table is for 101 df to infinity and
   //   is the value of the Z statistic (Normal).


//-- constructor --//
SimEnvGlobalStats::SimEnvGlobalStats( StatSimEnv *_pStatSimEnv )
//   : repNum( 0 ),
   : pStatSimEnv( _pStatSimEnv ),
     distLength( 0 ),
     obsLength( 0 ),
     timeLength( 0 ),
     continLength( 0 ),
     countLength( 0 ),

     distRow( 0 ),
     continRow( 0 ),
     timeRow( 0 ),
     obsRow( 0 ),
     countRow( 0 ),
     seedRow( 0 ),

     distObs( NULL ),
     initialSeed( NULL ),
     lastSeed( NULL ),

     obsObs( NULL ),
     obsMax( NULL ),
     obsMin( NULL ),
     obsAve( NULL ),
     obsMean( NULL ),
     obsStDev( NULL ),
     obsHalfWidth( NULL ),

     timeObs( NULL ),
     timeMax( NULL ),
     timeMin( NULL ),
     timeAve( NULL ),
     timeMean( NULL ),
     timeStDev( NULL ),
     timeHalfWidth( NULL ),

     continObs( NULL ),
     continMax( NULL ),
     continMin( NULL ),
     continAve( NULL ),
     continMean( NULL ),
     continStDev( NULL ),
     continHalfWidth( NULL ),

     countObs( NULL )

     {
     }


//-- destructor --//
SimEnvGlobalStats::~SimEnvGlobalStats( void )
   {
   //-- delete allocated arrays --//

   if ( distObs )
      delete distObs;

   if ( initialSeed )
      delete initialSeed;

   if ( lastSeed )
      delete lastSeed;

   if ( obsObs )
```

```
    delete obsObs;

 if ( obsMax )
    delete obsMax;

 if ( obsMin )
    delete obsMin;

 if ( obsAve )
    delete obsAve;

 if ( obsMean )
    delete obsMean;

 if ( obsStDev )
    delete obsStDev;

 if ( obsHalfWidth )
    delete obsHalfWidth;

 if ( timeObs )
    delete timeObs;

 if ( timeMax )
    delete timeMax;

 if ( timeMin )
    delete timeMin;

 if ( timeAve )
    delete timeAve;

 if ( timeMean )
    delete timeMean;

 if ( timeStDev )
    delete timeStDev;

 if ( timeHalfWidth )
    delete timeHalfWidth;

 if ( continObs )
    delete continObs;

 if ( continMax )
    delete continMax;

 if ( continMin )
    delete continMin;

 if ( continAve )
    delete continAve;

 if ( continMean )
    delete continMean;

 if ( continStDev )
    delete continStDev;

 if ( continHalfWidth )
    delete continHalfWidth;

 if ( countObs )
    delete countObs;
 }


float SimEnvGlobalStats::Get_T_Statistic( UINT numReps )
 {
 if ( numReps <= 2 )
    return tTable[ 0 ];

 else if ( numReps >= 102 )
    return tTable[ 100 ];

 else
    return tTable[ numReps - 2 ];
 }


float SimEnvGlobalStats::ComputeHalfWidth( float *arrayOfMeans, UINT lengthOf/
 {

 float standardDeviation = ComputeStDev( arrayOfMeans, lengthOfArray );

 float hWidth = Get_T_Statistic( lengthOfArray ) * standardDeviation /
                sqrt( (float)lengthOfArray );

 return hWidth;
 }

float SimEnvGlobalStats::ComputeMean( float *arrayOfMeans, UINT lengthOfArray
 {
 float sumMean = 0;

 for ( UINT i = 0; i < lengthOfArray; i++ )
    sumMean += arrayOfMeans[ i ];

 return ( sumMean / (float)lengthOfArray );
 }

float SimEnvGlobalStats::ComputeStDev( float *arrayOfMeans, UINT lengthOfArra
 {
 float sumSqrMeanValDiff = 0;
 float mean = ComputeMean( arrayOfMeans, lengthOfArray );

 for ( UINT i = 0; i < lengthOfArray; i++ )
    sumSqrMeanValDiff += ( arrayOfMeans[ i ] - mean ) *
                         ( arrayOfMeans[ i ] - mean );

 float oneOverDF = 1 / ( (float)lengthOfArray - 1 );

 return ( sqrt( oneOverDF * sumSqrMeanValDiff ) );
 }


void SimEnvGlobalStats::Setup( UINT _numReps, UINT _distLength,   // )
     UINT _obsLength, UINT _timeLength, UINT _continLength,
     UINT _countLength )
```

```
{
//-- store the lengths of the arrays --//

distLength       = _distLength;
obsLength        = _obsLength;
timeLength       = _timeLength;
continLength     = _continLength;
countLength      = _countLength;

//-- create the arrays first --//

if ( distLength )
   {
   distObs  = new long[ distLength ];
   initialSeed = new long[ distLength ];

   for ( UINT i = 0; i < distLength; i++ )
      {
      distObs[ i ] = 0;
      initialSeed[ i ] = 0;
      }

   lastSeed = new long[ distLength ];
   }

if ( obsLength )
   {
   obsObs = new long[ obsLength ];
   obsMax = new float[ obsLength ];
   obsMin = new float[ obsLength ];
   for ( UINT i = 0; i < obsLength; i++ )
      {
      obsObs[ i ] = 0;
      obsMax[ i ] = -1e37;
      obsMin[ i ] = 1e37;
      }

   obsAve = new TMatrix < float > ( obsLength, _numReps );
//   obsAve = new float[ obsLength ][ 100 ];

   //-- these don't need to be made yet, and it might be better to
   //   calculate on the fly
   obsMean = new float[ obsLength ];
   obsStDev = new float[ obsLength ];
   obsHalfWidth = new float[ obsLength ];

   }

if ( timeLength )
   {
   timeObs = new long[ timeLength ];
   timeMax = new float[ timeLength ];
   timeMin = new float[ timeLength ];
   for ( UINT i = 0; i < timeLength; i++ )
      {
      timeObs[ i ] = 0;
      timeMax[ i ] = -1e37;
      timeMin[ i ] = 1e37;
      }

   timeAve = new TMatrix < float > ( timeLength, _numReps );
//   timeAve = new float[ timeLength ][ 100 ];

   //-- these don't need to be made yet, and it might be better to
   //   calculate on the fly
   timeMean = new float[ timeLength ];
   timeStDev = new float[ timeLength ];
   timeHalfWidth = new float[ timeLength ];

   }

if ( continLength )
   {
   continObs = new  long[ continLength ];
   continMax = new float[ continLength ];
   continMin = new float[ continLength ];
   for ( UINT i = 0; i < continLength; i++ )
      {
      continObs[ i ] = 0;
      continMax[ i ] = -1e37;
      continMin[ i ] = 1e37;
      }

   continAve = new TMatrix < float > ( continLength, _numReps );
//   continAve = new float[ continLength ][ 100 ];

   //-- these don't need to be made yet, and it might be better to
   //   calculate on the fly
   continMean = new float[ continLength ];
   continStDev = new float[ continLength ];
   continHalfWidth = new float[ continLength ];

   }

if ( countLength )
   {
   countObs = new long[ countLength ];
   for ( UINT i = 0; i < countLength; i++ )
      countObs[ i ] = 0;
   }

}


void SimEnvGlobalStats::SetUpGlobalStatsOutput( UINT _numReps )
{
//-- generate the mean of the means --//
if ( obsLength )
   {
   float *tempArray = new float[ _numReps ];

   for ( UINT i = 0; i < obsLength; i ++ )
      {
      for ( UINT j = 0; j < _numReps; j++ )
         tempArray[ j ] = obsAve->Get( i, j );

      obsMean[ i ] =  ComputeMean( tempArray, _numReps );
      obsStDev[ i ] = ComputeStDev( tempArray, _numReps );
      obsHalfWidth[ i ] = ComputeHalfWidth( tempArray, _numReps );
      }
```

```
      delete tempArray;
      }

   //-- generate the standard deviation from the means --//
   if ( timeLength )
      {
      float *tempArray = new float[ _numReps ];

      for ( UINT i = 0; i < timeLength; i ++ )
         {
         for ( UINT j = 0; j < _numReps; j++ )
            tempArray[ j ] = timeAve->Get( i, j );

         timeMean[ i ] =  ComputeMean( tempArray, _numReps );
         timeStDev[ i ] = ComputeStDev( tempArray, _numReps );
         timeHalfWidth[ i ] = ComputeHalfWidth( tempArray, _numReps );
         }

      delete tempArray;
      }

   //-- generate the half-width used for the confidence interval --//
   if ( continLength )
      {
      float *tempArray = new float[ _numReps ];

      for ( UINT i = 0; i < continLength; i ++ )
         {
         for ( UINT j = 0; j < _numReps; j++ )
            tempArray[ j ] = continAve->Get( i, j );

         continMean[ i ] =  ComputeMean( tempArray, _numReps );
         continStDev[ i ] = ComputeStDev( tempArray, _numReps );
         continHalfWidth[ i ] = ComputeHalfWidth( tempArray, _numReps );
         }

      delete tempArray;
      }

   return;
   }



void SimEnvGlobalStats::SetGlobalDistStats( DistArray *pDistArray )
   {
   UINT length = pDistArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      Rand *pRand = pDistArray->Get( i );
      distObs [ distRow ] += pRand->GetObservations();
      lastSeed[ distRow ] = pRand->GetSeed();
      distRow++;
      }

   return;
   }


//-- stores the initial seeds in the global stats object
//   to be retrieved when reports are generated
void SimEnvGlobalStats::SetInitialSeeds( DistArray *pDistArray )
   {
   static UINT initialSeedRow = 0;

   UINT length = pDistArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      Rand *pRand = pDistArray->Get( i );
      long _initialSeed = pRand->GetInitialSeed( );

      initialSeed[ initialSeedRow ] = _initialSeed;
      initialSeedRow++;
      }

   return;
   }


void SimEnvGlobalStats::SetGlobalDistSeeds( DistArray *pDistArray )
   {
   UINT length = pDistArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      Rand *pRand = pDistArray->Get( i );
      pRand->SetSeed( lastSeed[ seedRow ] );
      seedRow++;
      }

   return;
   }


void SimEnvGlobalStats::SetContinMax( float maxValue )
   {
   float currentMaxValue = continMax[ continRow ];

   if ( maxValue > currentMaxValue )
      continMax[ continRow ] = maxValue;

   return;
   }

void SimEnvGlobalStats::SetContinMin( float minValue )
   {
   float currentMinValue = continMin[ continRow ];

   if ( minValue < currentMinValue )
      continMin[ continRow ] = minValue;

   return;
   }
```

```
void SimEnvGlobalStats::SetGlobalContinStats( StatsContinArray *pContinArray )
   {
   UINT length = pContinArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      StatsContin *pContin    = pContinArray->Get( i );
      continObs[ continRow ]     += pContin->GetObservations();
      SetContinMax( pContin->GetMaximum() );
      SetContinMin( pContin->GetMinimum() );
      continAve->Set( continRow, pStatSimEnv->repNum, pContin->GetMean() );
      continRow++;
      }

   return;
   }

void SimEnvGlobalStats::SetTimeMax( float maxValue )
   {
   float currentMaxValue = timeMax[ timeRow ];

   if ( maxValue > currentMaxValue )
      timeMax[ timeRow ] = maxValue;

   return;
   }

void SimEnvGlobalStats::SetTimeMin( float minValue )
   {
   float currentMinValue = timeMin[ timeRow ];

   if ( minValue < currentMinValue )
      timeMin[ timeRow ] = minValue;

   return;
   }


void SimEnvGlobalStats::SetGlobalTimeStats( StatsTimeArray *pTimeArray )
   {
   UINT length = pTimeArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      StatsTime *pTime          = pTimeArray->Get( i );
      timeObs[ timeRow ]        += pTime->GetObservations();
      SetTimeMax( pTime->GetMaximum() );
      SetTimeMin( pTime->GetMinimum() );
      timeAve->Set( timeRow, pStatSimEnv->repNum, pTime->GetMean() );
      timeRow++;
      }

   return;
   }


void SimEnvGlobalStats::SetObsMax( float maxValue )
   {
   float currentMaxValue = obsMax[ obsRow ];

   if ( maxValue > currentMaxValue )
      obsMax[ obsRow ] = maxValue;

   return;
   }

void SimEnvGlobalStats::SetObsMin( float minValue )
   {
   float currentMinValue = obsMin[ obsRow ];

   if ( minValue < currentMinValue )
      obsMin[ obsRow ] = minValue;

   return;
   }


void SimEnvGlobalStats::SetGlobalObsStats( StatsObsArray *pObsArray )
   {
   UINT length = pObsArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      StatsObs *pObs                 = pObsArray->Get( i );
      obsObs[ obsRow ]              += pObs->GetObservations();
      SetObsMax( pObs->GetMaximum() );
      SetObsMin( pObs->GetMinimum() );
      obsAve->Set( obsRow, pStatSimEnv->repNum, pObs->GetMean() );
      obsRow++;
      }

   return;
   }


void SimEnvGlobalStats::SetGlobalCountStats( CounterArray *pCountArray )
   {
   UINT length = pCountArray->Length();

   for ( UINT i = 0; i < length; i++ )
      {
      Counter *pCount             = pCountArray->Get( i );
      countObs[ countRow ]       += pCount->GetObservations();
      countRow++;
      }

   return;
   }
```

Appendix K: Runge-Kutta integrators

```
Print date: 8/17/1992     Print time:  8:15
//---------------------------------------------------------------------
//  PROGRAM: integrat.cpp
//
//  PURPOSE: Integration methods
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//------------------------ Includes ----------------------------//

#include <simenv.hpp>


//--------------------- Integration Methods --------------------//


//-- IntRKS() ---------------------------------------------------
//
//--- general purpose Runge-Kutta integrator (fourth order)
//    code is designed and modified from "Applied Numerical Methods" by
//    Brice Carnahan, H.A. Luther, and James O. Wilkes., 1969., pgs 374-375.
//
//--INPUTS
//   int    svCount    = state variable count
//   float  stateVar   = state variable value
//   double derivative = derivative value
//   float  *time      = time variable address
//   float  dt         = timestep
//---------------------------------------------------------------------

/*
---------------------------------------------------------------------
Example SimObj::State() method

void MySimObj::State( int svCount, double* stateVar,
   double* derivative, double time )
   {
   sv02 = stateVar[ 0  ];
   etc....

   d02_dt = ... etc.
   .. etc...

   // derivative[ 0 ] = d02_dt;
   // ...etc...
   }
---------------------------------------------------------------------
*/

BOOL IntRKS(                        // )
      SimObj *pSimObj,
      double timeStep,
      double currentTime,
      int    svCount,
      double *stateVar,             // state variable array
      STATEPROC lpFn
      )
  {
```

```
const int totalPasses = 4;
int  i;                         // loop counter

double *derivative = new double[ svCount ];
double *svTemp     = new double[ svCount ];
double *phi        = new double[ svCount ];      // increment function

for ( int pass = 0; pass < totalPasses; pass++ )
   {
   //-- get derivative and state values from object --//
   if ( lpFn )
      (pSimObj->*lpFn)( svCount, stateVar, derivative, currentTime );
   else
      pSimObj->State( svCount, stateVar, derivative, currentTime );

   //-- select appropriate RKS stage --//
   switch ( pass )
      {
      case 0:
         for ( i=0; i < svCount; i++ )
            {
            svTemp[ i ]   = stateVar[ i ];
            phi[ i ]      = derivative[ i ];
            stateVar[ i ] = svTemp[ i ] + \
                                ( 0.5 * timeStep * derivative[ i ] );
            }

         //-- increment temporary time variable --//
         currentTime += ( 0.5 * timeStep );
         break;

      case 1:
         for ( i=0; i < svCount; i++ )
            {
            phi[i]       += ( 2.0 * derivative[ i ] );
            stateVar[ i ] = svTemp[ i ] + \
                              ( 0.5 * timeStep * derivative[ i ] );
            }
         break;

      case 2:
         for ( i=0; i < svCount; i++ )
            {
            phi[i]     += ( 2.0 * derivative[ i ] );
            stateVar[i] = svTemp[ i ] + ( timeStep * derivative[ i ] );
            }

         //-- increment temporary time variable --//
         currentTime += ( 0.5 * timeStep );
         break;

      case 3:
         for ( i=0; i < svCount; i++ )
            {
            stateVar[i] = svTemp[i] + \
                    ( (phi[ i ] + derivative[ i ]) * timeStep / 6.0 );
            }
         break;

      } // end of pass switch
```

```
    ) // end of for loop

  //-- clean up --//
  delete derivative;
  delete svTemp;
  delete phi;

  return TRUE;
  )


/*
-----------------------------------------------------------------------
THE NEXT INTEGRATOR TO IMPLEMENT!

Runge-Kutta-Fehlberg Formula  (RKF45) 1969
from "An Introduction to Numerical Computations by Sidney Yakowitz and
Ferenc Szidarovszky 1986. pg. 324.

Based on the six seperate calls to subroutine State, (RKF45) computes
an estimate for each state variable for one timeStep, an estimate of
the error for this step, and a new proposed size for the next step.

The step is usually specified with minimum and maximum values. (RKF45)
tries to use the largest stepsize, but the next proposed size may be
smaller in order to meet the minimum desired accuracy for the state
variables.

If the error is too large, the integration is repeated with a smaller
stepsize.
-----------------------------------------------------------------------
*/
```

Appendix L: Output report methods

```
Print date: 8/17/1992     Print time:  6:01
//-------------------------------------------------------------------
//  PROGRAM: report.hpp
//
//  PURPOSE: Output report functions
//-------------------------------------------------------------------

#if !defined _REPORT_HPP
#define      _REPORT_HPP 1

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _SIMOBJ_HPP )
#include <simobj.hpp>
#endif



void MyHeaderOutput( SimEnv *pSimEnv );
void MyGlobalHeaderOutput( StatSimEnv *pSimEnv );

int MyDistOutput( REPORT &report );
int MyObsOutput  ( REPORT &report );
int MyGlobalObsOutput ( REPORT &report );
int MyCountOutput( REPORT &report );

void PrintDistTitle( char *buffer );
void PrintDistHeader( char *buffer );
void PrintGlobalDistHeader( char *buffer );

void PrintObsTitle( char *buffer );
void PrintObsHeader( char *buffer );
void PrintGlobalObsHeader( char *buffer );

void PrintTimeTitle( char *buffer );

void PrintContinTitle( char *buffer );

void PrintCountTitle( char *buffer );
void PrintCountHeader( char *buffer );
void PrintGlobalCountHeader( char *buffer );

void PrintSingleRunTitle( char *buffer );
void PrintReportHeader( SimEnv *pSimEnv, char *buffer );

void PrintMultipleRunTitle( StatSimEnv *pSimEnv, char *buffer );



int TraceOutput( SimEnv*, SimObj*, UINT, TraceArray* );
void ViewEventList( SimObj *pSimObj, double eventTime, UINT priority,
                    UINT eventType );


#endif
```

```
Print date: 8/17/1992     Print time:  6:07
//-------------------------------------------------------------------
// PROGRAM: report.cpp
//-------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _REPORT_HPP )
#include <report.hpp>
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _SIMGLOBA_HPP )
#include <simgloba.hpp>
#endif


extern "C"
{
#include <iostream.h>
#include <stdio.h>
}


#if defined ( _Windows ) && !defined ( EASYWIN )

#include <logwnd.hpp>
extern LogWindow *gpTraceWnd;
extern LogWindow *gpLogWnd;
#define TRACEOUT (*gpTraceWnd)
#define LOGOUT   (*gpLogWnd)

#else

#define TRACEOUT cout
#define LOGOUT   cout

#endif

//-------------------------- Constants --------------------------//

const int REPORT_BUFFER = 256;


/*
---------------------------------------------------------------------
Report output format.

Header output: "Single run statistics"
               Simulation: char*
               Start time: char*
               End time  : char*
               Statistical reset time: char*
```

```
( PMMCG )                          DISTRIBUTIONS
Xi = a*(Xi-1) mod m

Object  Statistic  Distribution  A     B     C     Seed     #Obs   Multiplier
-------------------------------------------------------------------------------
char*,  char*,     char*,        float,float,float,long,  long,  long


                        OBSERVATIONAL VARIABLES

Object  Statistic  Average  Variation  Min      Max      #Obs  Final Value
-------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float


                     DISCRETE TIME BASED VARIABLES

Object  Statistic  Average  Variation  Min      Max      #Obs  Final Value
-------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float


                   CONTINUOUS TIME BASED VARIABLES

Object  Statistic  Average  Variation  Min      Max      #Obs  Final Value
-------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float


                          COUNT VARIABLES

Object  Statistic  Count                                    #Obs
-------------------------------------------------------------------
char*,  char*,     long,                                    long
-------------------------------------------------------------------


-------------------------------------------------------------------
Report output format.

Header output: "Mutiple run statistics"
               Simulation: char*
               Average run length: char*
               Statistical reset time: char*
               Number of runs: char*

( PMMCG )                          DISTRIBUTIONS
Xi = a*(Xi-1) mod m

Object  Statistic  Distribution  A     B     C     Seed     #Obs   Multiplier
-------------------------------------------------------------------------------
char*,  char*,     char*,        float,float,float,long,  long,  long
```

```
                    OBSERVATIONAL VARIABLES

                                                                    Lower Upper
                                    Standard                          CL
Object  Statistic  Average  deviation  Min      Max      #Obs  95%   95%
-------------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float,float

                    DISCRETE TIME BASED VARIABLES

                                                                    Lower Upper
                                    Standard                          CL
Object  Statistic  Average  deviation  Min      Max      #Obs  95%   95%
-------------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float,float

                  CONTINUOUS TIME BASED VARIABLES

                                                                    Lower Upper
                                    Standard                          CL
Object  Statistic  Average  deviation  Min      Max      #Obs  95%   95%
-------------------------------------------------------------------------------
char*,  char*,     float,   float,     float,   float,   long, float,float

                      COUNT VARIABLES

Object  Statistic  Count                                  #Obs
-------------------------------------------------------------------
char*,  char*,     long,                                  long

-------------------------------------------------------------------

*/

void MyHeaderOutput( SimEnv *pSimEnv )
   {
   char buffer[ REPORT_BUFFER ];

   if ( ! pSimEnv->GlobalStatsAvailable( ) )
      {
      PrintSingleRunTitle( buffer );
      PrintReportHeader( pSimEnv, buffer );
      }

   //-- print distribution information --//
   if ( pSimEnv->DistStatsAvailable() )
      {
      PrintDistTitle( buffer );
      PrintDistHeader( buffer );

      pSimEnv->EnumDistStats( MyDistOutput );
      }

   //-- print observational statistics --/
   if ( pSimEnv->ObsStatsAvailable() )
      {
      PrintObsTitle( buffer );
      PrintObsHeader( buffer );

      pSimEnv->EnumObsStats( MyObsOutput );
      }

   //-- print time based statistics --//
   if( pSimEnv->ContinStatsAvailable() )
      {
      PrintContinTitle( buffer );
      PrintObsHeader( buffer );

      pSimEnv->EnumContinStats( MyObsOutput );
      }

   //-- print time based statistics --//
   if( pSimEnv->TimeStatsAvailable() )
      {
      PrintTimeTitle( buffer );
      PrintObsHeader( buffer );

      pSimEnv->EnumTimeStats( MyObsOutput );
      }

   //-- print counters --//
   if( pSimEnv->CountStatsAvailable() )
      {
      PrintCountTitle( buffer );
      PrintCountHeader( buffer );

      pSimEnv->EnumCountStats( MyCountOutput );
      }

   }


void MyGlobalHeaderOutput( StatSimEnv *pSimEnv )
   {
   char buffer[ REPORT_BUFFER ];

   if ( pSimEnv->GlobalStatsAvailable( ) )
      {
      PrintMultipleRunTitle( pSimEnv, buffer );
      PrintReportHeader( pSimEnv, buffer );
      }

   //-- print observational statistics --/
   if ( pSimEnv->GlobalStatsAvailable() )
      {
      SimEnvGlobalStats *pGlobalStats = pSimEnv->GetGlobalStatsPtr();

      if ( pGlobalStats->GetDistLength() )
         {
         PrintDistTitle( buffer );
         PrintGlobalDistHeader( buffer );
         pSimEnv->EnumGlobalDistStats( MyDistOutput );
         }

      if ( pGlobalStats->GetObsLength() )
         {
         PrintObsTitle( buffer );
         PrintGlobalObsHeader( buffer );
         pSimEnv->EnumGlobalObsStats( MyGlobalObsOutput );
         }
```

```
      if ( pGlobalStats->GetTimeLength() )
         {
         PrintTimeTitle( buffer );
         PrintGlobalObsHeader( buffer );
         pSimEnv->EnumGlobalTimeStats( MyGlobalObsOutput );
         }

      if ( pGlobalStats->GetContinLength() )
         {
         PrintContinTitle( buffer );
         PrintGlobalObsHeader( buffer );
         pSimEnv->EnumGlobalContinStats( MyGlobalObsOutput );
         }

      if ( pGlobalStats->GetCountLength() )
         {
         PrintCountTitle( buffer );
         PrintGlobalCountHeader( buffer );
         pSimEnv->EnumGlobalCountStats( MyCountOutput );
         }

      ) // end of IF

   return;
   }


int MyDistOutput( REPORT &report )
   {
   char buffer[ REPORT_BUFFER ];

   sprintf( buffer, "%-9.8s %-10.9s %-10.9s %-6.5s %-6.5s %-6.5s %-10lu %-5lu %-
      report.simObjName, report.idName, report.distName, report.paramA,
      report.paramB, report.paramC, report.initialSeed, report.observations,
      report.multiplier );

   LOGOUT << buffer;

   return 0;
   }


int MyObsOutput( REPORT &report )
   {
   char buffer[ REPORT_BUFFER ];

   sprintf( buffer,  "%-9.8s %-10.9s %-10.4f %-10.4f %-8.4f %-11.4f %-5lu %-12.4
         report.simObjName, report.idName, report.average,
         report.variation, report.minimum,
         report.maximum, report.observations, report.finalValue );
   LOGOUT << buffer;

   return 0;
   }


int MyGlobalObsOutput( REPORT &report )
   {
   char buffer[ REPORT_BUFFER ];

   sprintf( buffer,  "%-9.8s %-10.9s %-10.4f %-10.4f %-8.4f %-8.4f %5lu %-12.4
         report.simObjName, report.idName, report.average,
         report.sDeviation, report.minimum, report.maximum,
         report.observations, report.LCL, report.UCL );
   LOGOUT << buffer;

   return 0;
   }

int MyCountOutput( REPORT &report )
   {
   char buffer[ REPORT_BUFFER ];

   sprintf( buffer,  "%-9.8s %-10.9s %-5lu
   report.simObjName, report.idName, report.count, report.observations );
   LOGOUT << buffer;

   return 0;
   }

void PrintSingleRunTitle( char *buffer )
   {
   sprintf( buffer, "\n            Single Run Statistics \n" );
   LOGOUT << buffer;

   return;
   }

void PrintMultipleRunTitle( StatSimEnv *pSimEnv, char *buffer )
   {
   sprintf( buffer, "\n            Multiple Run Global Statistics \n" );
   LOGOUT << buffer;

   sprintf( buffer, "              Replication Method              \n");
   LOGOUT << buffer;

   sprintf( buffer, "\nNumber of replications (obs): %u\n",
         pSimEnv->GetNumReps() );

   LOGOUT << buffer;

   return;
   }

void PrintReportHeader( SimEnv *pSimEnv, char *buffer )
   {
   //-- print out the header --//
   sprintf( buffer, "\nSimulation: %s\n", pSimEnv->GetName() );
   LOGOUT << buffer;

   sprintf( buffer, "Start time: %f\n", (float) pSimEnv->GetStartTime() );
   LOGOUT << buffer;

   sprintf( buffer, "End time: %f\n", (float) pSimEnv->GetCurrentTime() );
   LOGOUT << buffer;

   sprintf( buffer, "Statistical reset time: %f\n",
       (float) pSimEnv->GetLastClearStatsTime() );
```

```
    LOGOUT << buffer;

    return;
    }

void PrintDistTitle( char *buffer )
    {
    sprintf( buffer,  "\n\n( PMMCG )                    DISTRIBUTIONS\n" );
    LOGOUT << buffer;

    sprintf( buffer, "Xi = a*(Xi-1) mod m \n" );
    LOGOUT << buffer;

    return;
    }

void PrintDistHeader( char *buffer )
    {
    sprintf( buffer,  "Object    Statistic  Dist       A      B      C      Seed
    LOGOUT << buffer;

    sprintf( buffer,  "-------------------------------------------------------
    LOGOUT << buffer;

    return;
    }

void PrintGlobalDistHeader( char *buffer )
    {
    sprintf( buffer,  "
    LOGOUT << buffer;

    sprintf( buffer,  "Object    Statistic  Dist       A      B      C      Seed
    LOGOUT << buffer;

    sprintf( buffer,  "-------------------------------------------------------
    LOGOUT << buffer;

    return;
    }

void PrintObsTitle( char *buffer )
    {
    sprintf( buffer,  "\n\n                          OBSERVATIONAL VARIABLES \n
    LOGOUT << buffer;

    return;
    }

void PrintObsHeader( char *buffer )
    {
    sprintf( buffer,  "
    LOGOUT << buffer;

    sprintf( buffer,  "Object    Statistic  Average    Variation  Min      Max
    LOGOUT << buffer;

    sprintf( buffer,  "-------------------------------------------------------
    LOGOUT << buffer;

    return;
    }

void PrintGlobalObsHeader( char *buffer )
    {
    sprintf( buffer,  "                              Standard
    LOGOUT << buffer;

    sprintf( buffer,  "Object    Statistic  Average  deviation  Min      Max
    LOGOUT << buffer;

    sprintf( buffer,  "-------------------------------------------------------
    LOGOUT << buffer;

    return;
    }

void PrintTimeTitle( char *buffer )
    {
    sprintf( buffer,  "\n\n                          DISCRETE TIME BASED VAR
    LOGOUT << buffer;

    return;
    }

void PrintContinTitle( char *buffer )
    {
    sprintf( buffer,  "\n\n                          CONTINUOUS TIME BASED V
    LOGOUT << buffer;

    return;
    }

void PrintCountTitle( char *buffer )
    {
    sprintf( buffer,  "\n\n                          COUNT VARIABLES \n\n" )
    LOGOUT << buffer;

    return;
    }

void PrintCountHeader( char *buffer )
    {
    sprintf( buffer,  "Object    Statistic  Count
    LOGOUT << buffer;

    sprintf( buffer,  "-------------------------------------------------------
    LOGOUT << buffer;

    return;
    }

void PrintGlobalCountHeader( char *buffer )
    {
    sprintf( buffer,  "
    LOGOUT << buffer;

    sprintf( buffer,  "Object    Statistic  Count
    LOGOUT << buffer;
```

```
   sprintf( buffer,  "-------------------------------------------------------
   LOGOUT << buffer;

   return;
   }



//-- TraceOutput() ------------------------------------------------
//
//-- trace information on each SimObject. Useful for debugging.
//-----------------------------------------------------------------

int TraceOutput( SimEnv *pSimEnv, SimObj *pSimObj, UINT eventType, \
       TraceArray *pTraceArray )
   {
   if ( ! pTraceArray )
      return 0;

   if ( pTraceArray->Length() == 0 )
      return 0;

   char buffer[ REPORT_BUFFER ];

   UINT length = pTraceArray->Length();

   sprintf( buffer,  "\\012\\012TRACE -- Time: %-20.5g",
                  (float) pSimEnv->GetCurrentTime() );

   TRACEOUT << buffer;

   sprintf( buffer,  "SimObj: %-20.20s Priority: %-4d Event: %-5u\n",
           pSimObj->GetName(),
           (int) pSimObj->GetPriority(),
           (int) eventType );

   TRACEOUT << buffer;

   sprintf( buffer,  "Data:                Value:\n");
   TRACEOUT << buffer;

   sprintf( buffer,  "-------------------------------------------------------\
   TRACEOUT << buffer;

   double value = pTraceArray->GetValue( 0 );

   sprintf( buffer,  "%-20.20s %-20.5g\n", \
           (*pTraceArray)[ 0 ]->name, value );


   TRACEOUT << buffer;

   for( UINT i = 1; i < length; i++ )
      {
      value = pTraceArray->GetValue( i );

      sprintf( buffer,  "%-20.20s %-20.5g\n", pTraceArray->Get(i)->name, value )
      TRACEOUT << buffer;
      }
```

```
   return 0;
   }


//-- ViewEventList() ----------------------------------------------
//
//-- Generic application function for displaying the event list...
//-- DOS programs use standard output
//-- Windows programs use the LogWindow
//-----------------------------------------------------------------

void ViewEventList( SimObj *pSimObj, double eventTime, UINT priority,
                    UINT eventType )
   {
   char buffer[ OUTPUTBUFFER ];

   if ( pSimObj == NULL )
      {
      if ( eventType == 0x0000 )     // start of listing
         {
         sprintf( buffer, "\nObject  EventTime  Priority  EventType\n" );
         LOGOUT << buffer;
         return;
         }

      else if ( eventType == ET_ENUMEVENTLIST )  // end of listing
         {
         LOGOUT << "\n";
         return;
         }
      }

   //-- non-null pSimObj, so display event info --//

   char name [ 256 ];
   if ( pSimObj )
      strcpy( name, pSimObj->GetName() );
   else
      strcpy( name, "System Event" );

   sprintf( buffer, "%-10.9s %-12.4f %-4d %-5d\n",
         name, eventTime, priority, eventType );
   LOGOUT << buffer;

   return;
   }



/*
-------------------------------------
void GetTraceData( TYPE typeFlag, void *pData, char *dataString )
   {
   switch( typeFlag )
      {
      // case TYPE_UNSIGNED_CHAR:
      // case TYPE_CHAR:

      case TYPE_ENUM:
```

```
            sprintf( dataString, %-u, *((UINT*) pData) );
            break;

         case TYPE_UNSIGNED_INT:
            sprintf( dataString, %-u, *((UINT*) pData) );
            break;

         // case TYPE_SHORT_INT:

         case TYPE_INT:
            sprintf( dataString, %-d, *((int*) pData) );
            break;

         // case TYPE_UNSIGNED_LONG:

         case TYPE_LONG:
            sprintf( dataString, %-dl, *((LONG*) pData) );
            break;

         case TYPE_FLOAT:
            sprintf( dataString, %-f, *((float*) pData) );
            break;

         case TYPE_DOUBLE:
            sprintf( dataString, %-fL, *((double*) pData) );
            break;

         //case TYPE_LONG_DOUBLE:
         //   sprintf( dataString, %-fL, *((double*) pData) );
         //  break;

         case default:
            sprintf( dataString, %-d, *((int*) pData) );
            break;

         )  // end of SWITCH

      return dataString;
      )
-------------------------------------
*/
```

```
Print date: 8/17/1992     Print time:  8:06
//---------------------------------------------------------------------
//  PROGRAM: se_rep.cpp
//
//  PURPOSE: report methods for the simulation environment
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//------------------------- Includes --------------------------//

#if !defined ( _SIMOBJ_HPP )
#include <simobj.hpp>
#endif

#if !defined ( _SIMENV_HPP )
#include <simenv.hpp>
#endif

#if !defined ( _ST_ARRAY_HPP )
#include <st_array.hpp>
#endif

#if !defined ( _RANDOM_HPP )
#include <random.hpp>
#endif


//-- send distribution info to user defined output function --//
void SimEnv::EnumDistStats( REPORTPROC lpReportFn )
   {
   char buffer[13];
   REPORT report;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      DistArray *pDistArray = \
            (DistArray*) pSimObj->IsDistsOn();

      if( pDistArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pDistArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            Rand *pRand = (Rand*) pDistArray->Get( j );

            strcpy( report.idName, pRand->GetName() );
            strcpy( report.distName, pRand->GetDistName() );
            strcpy( report.paramA, pRand->GetA() );
            strcpy( report.paramB, pRand->GetB() );
            strcpy( report.paramC, pRand->GetC() );
            report.initialSeed         = pRand->GetInitialSeed();
            report.observations = pRand->GetObservations();
            report.multiplier   = pRand->GetMultiplier();

            lpReportFn( report );
            } // end of inner FOR

         } // end of IF

      } // end of outer FOR

   return;
   }


//-- send observational based statistics to user defined output function --//
void SimEnv::EnumObsStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsObsArray *pStatsArray = \
            (StatsObsArray*) pSimObj->IsObsStatsOn();


      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsObs *pStats = (StatsObs*) pStatsArray->Get( j );

            strcpy( report.idName, pStats->GetName() );
            report.average      = (float) pStats->GetMean();
            report.variation    = (float) pStats->GetVariation();
            report.minimum      = (float) pStats->GetMinimum();
            report.maximum      = (float) pStats->GetMaximum();
            report.observations =         pStats->GetObservations();
            report.finalValue   = (float) pStats->GetValue();

            lpReportFn( report );
            } // end of inner FOR

         } // end of IF

      } // end of outer FOR

   return;
   }
```

```
//-- send time based statistics to user defined output function --//
void SimEnv::EnumContinStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsContinArray *pStatsArray = \
            (StatsContinArray*) pSimObj->IsContinStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsContin *pStats = \
                     (StatsContin*) pStatsArray->Get( j );
            strcpy( report.idName, pStats->GetName() );
            report.average      = (float) pStats->GetMean();
            report.variation    = (float) pStats->GetVariation();
            report.minimum      = (float) pStats->GetMinimum();
            report.maximum      = (float) pStats->GetMaximum();
            report.observations =         pStats->GetObservations();
            report.finalValue   = (float) pStats->GetValue();

            lpReportFn( report );
            } // end of inner FOR

         }  // end of IF

      }  // end of outer FOR

   return;

   }


//-- send time based statistics to user defined output function --//
void SimEnv::EnumTimeStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsTimeArray *pStatsArray = \
            (StatsTimeArray*) pSimObj->IsTimeStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsTime *pStats = (StatsTime*) pStatsArray->Get( j );
            strcpy( report.idName, pStats->GetName() );
            report.average      = (float) pStats->GetMean();
            report.variation    = (float) pStats->GetVariation();
            report.minimum      = (float) pStats->GetMinimum();
            report.maximum      = (float) pStats->GetMaximum();
            report.observations =         pStats->GetObservations();
            report.finalValue   = (float) pStats->GetValue();

            lpReportFn( report );
            } // end of inner FOR

         }  // end of IF

      }  // end of outer FOR

   return;

   }


//-- send count based statistics to user defined output function --//
void SimEnv::EnumCountStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      CounterArray *pStatsArray = \
            (CounterArray*) pSimObj->IsCountStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            Counter *pStats = pStatsArray->Get( j );

            strcpy( report.idName, pStats->GetName() );
            report.count        = pStats->GetCount();
            report.observations = pStats->GetObservations();
```

```
            lpReportFn( report );

            ) // end of inner FOR

          )  // end of IF

        )  // end of outer FOR

    return;

    )


void SimEnv::GetSimObjName( SimObj *pSimObj, REPORT &report )
    {
    //-- get the actual simObjName --//
    strncpy( report.simObjName, pSimObj->GetName(), 12 );
    report.simObjName[ 12 ] = '\0';

    return;
    }
```

```
Print date: 8/17/1992     Print time:  8:06
//---------------------------------------------------------------------
//  PROGRAM: sse_rep.cpp
//---------------------------------------------------------------------

#include <windows.h>
#pragma hdrstop

//-------------------------- Includes --------------------------//

#if !defined ( _SIMOBJ_HPP )
#include <simobj.hpp>
#endif

#if !defined ( _STSIMENV_HPP )
#include <stsimenv.hpp>
#endif

#if !defined ( _ST_ARRAY_HPP )
#include <st_array.hpp>
#endif

#if !defined ( _RANDOM_HPP )
#include <random.hpp>
#endif


//-- send global distribution info to user defined output function --//
void StatSimEnv::EnumGlobalDistStats( int (*lpReportFn)( REPORT& ) )
   {
   char buffer[13];
   REPORT report;

   //-- global stats row to print --//
   UINT rowCount = 0;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      DistArray *pDistArray = \
            (DistArray*) pSimObj->IsDistsOn();

      if( pDistArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pDistArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            Rand *pRand = (Rand*) pDistArray->Get( j );

            strcpy( report.idName, pRand->GetName() );
            strcpy( report.distName, pRand->GetDistName() );
            strcpy( report.paramA, pRand->GetA() );
            strcpy( report.paramB, pRand->GetB() );
            strcpy( report.paramC, pRand->GetC() );

//-- need to get initialSeeds from the StatSimEnv !!

//            report.initialSeed  = pRand->GetInitialSeed();
            report.initialSeed  = pGlobalStats->GetInitialSeed( rowCount );

            report.observations = pGlobalStats->GetDistObs( rowCount ) /
                                  (long)numReps;
            report.multiplier   = pRand->GetMultiplier();

            lpReportFn( report );
            rowCount++;
            } // end of inner FOR

         } // end of IF

      } // end of outer FOR

   return;
   }


//-- send observational based statistics to user defined output function --//
void StatSimEnv::EnumGlobalObsStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- global stats row to print --//
   UINT rowCount = 0;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsObsArray *pStatsArray = \
            (StatsObsArray*) pSimObj->IsObsStatsOn();


      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsObs *pStats = (StatsObs*) pStatsArray->Get( j );

            strcpy( report.idName, pStats->GetName() );
            report.average      = pGlobalStats->GetObsMean( rowCount );
            report.sDeviation   = pGlobalStats->GetObsStDev( rowCount );
            report.minimum      = pGlobalStats->GetObsMin( rowCount );
```

```
            report.maximum      = pGlobalStats->GetObsMax( rowCount );
            report.observations = pGlobalStats->GetObsObs( rowCount ) /
                                  (long)numReps;
            report.halfWidth    = pGlobalStats->GetObsHalfWidth( rowCount );
            report.LCL    = (((float)(0) >
                             (float)(report.average - report.halfWidth)) ?
                             (float)(0) :
                             (float)(report.average - report.halfWidth));
//            report.LCL    = max( (float) 0, (float)(report.average - report.hal
            report.UCL    = report.average + report.halfWidth;

            lpReportFn( report );
            rowCount++;
            ) // end of inner FOR

         ) // end of IF

      ) // end of outer FOR

   return;
   )


//-- send time based statistics to user defined output function --//
void StatSimEnv::EnumGlobalContinStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- global stats row to print --//
   UINT rowCount = 0;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsContinArray *pStatsArray = \
            (StatsContinArray*) pSimObj->IsContinStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsContin *pStats = \
                     (StatsContin*) pStatsArray->Get( j );
            strcpy( report.idName, pStats->GetName() );
            report.average      = pGlobalStats->GetContinMean( rowCount );
            report.sDeviation   = pGlobalStats->GetContinStDev( rowCount );
            report.minimum      = pGlobalStats->GetContinMin( rowCount );
            report.maximum      = pGlobalStats->GetContinMax( rowCount );
            report.observations = pGlobalStats->GetContinObs( rowCount ) /
                                  (long)numReps;
            report.LCL    = (((float)(0) >
                             (float)(report.average - report.halfWidth)) ?
                             (float)(0) :
                             (float)(report.average - report.halfWidth));
//            report.LCL    = max( (float) 0, (float)(report.average - report.h
            report.UCL    = report.average + report.halfWidth;

            lpReportFn( report );
            rowCount++;
            } // end of inner FOR

         ) // end of IF

      ) // end of outer FOR

   return;

   )


//-- send time based statistics to user defined output function --//
void StatSimEnv::EnumGlobalTimeStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- global stats row to print --//
   UINT rowCount = 0;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      StatsTimeArray *pStatsArray = \
            (StatsTimeArray*) pSimObj->IsTimeStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            StatsTime *pStats = (StatsTime*) pStatsArray->Get( j );
            strcpy( report.idName, pStats->GetName() );
            report.average      = pGlobalStats->GetTimeMean( rowCount );
            report.sDeviation   = pGlobalStats->GetTimeStDev( rowCount );
            report.minimum      = pGlobalStats->GetTimeMin( rowCount );
            report.maximum      = pGlobalStats->GetTimeMax( rowCount );
            report.observations = pGlobalStats->GetTimeObs( rowCount ) /
                                  (long)numReps;
            report.halfWidth    = pGlobalStats->GetTimeHalfWidth( rowCount );
            report.LCL    = (((float)(0) >
                             (float)(report.average - report.halfWidth)) ?
                             (float)(0) :
                             (float)(report.average - report.halfWidth));
//            report.LCL          = max( (float) 0, report.average - report.h
            report.UCL          = report.average + report.halfWidth;
```

```
            lpReportFn( report );
            rowCount++;
            } // end of inner FOR

         } // end of IF

      } // end of outer FOR

   return;

   }


//-- send count based statistics to user defined output function --//
void StatSimEnv::EnumGlobalCountStats( REPORTPROC lpReportFn )
   {
   REPORT report;

   //-- global stats row to print --//
   UINT rowCount = 0;

   //-- goto head of SimObj list --//
   simObjList.GoToHead();

   for( UINT i = 0; i < simObjList.Length(); i++, ++simObjList )
      {
      SIMOBJ *pSimObjStruct = simObjList.Examine();
      SimObj *pSimObj = pSimObjStruct->pSimObj;

      CounterArray *pStatsArray = \
            (CounterArray*) pSimObj->IsCountStatsOn();

      if( pStatsArray )
         {
         GetSimObjName( pSimObj, report );

         UINT length = pStatsArray->Length();
         for( UINT j = 0; j < length; j++ )
            {
            Counter *pStats = pStatsArray->Get( j );

            strcpy( report.idName, pStats->GetName() );
            report.observations = pGlobalStats->GetCountObs( rowCount ) /
                                  (long) numReps;

            lpReportFn( report );
            rowCount++;
            } // end of inner FOR

         } // end of IF

      } // end of outer FOR

   return;

   }
```