Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

OBJECT ORIENTATION & THREE PHASE SIMULATION

Michael Pidd

The Management School
Lancaster University
Lancaster LA1 4YX
UK

ABSTRACT

Object oriented methods are now popular in many areas of computing, increasingly so in those areas associated with analysis and management science. The first system to incorporate many of the ideas of object orientation was SIMULA, which is also associated with a process-interaction view of discrete simulation. Possibly as a consequence of this history, most attempts to take an object oriented approach within discrete simulation seem to have been associated with a process-based or process interaction based modelling paradigm. This paper considers the advantages of object orientation and applies an object oriented approach to the provision of a three-phase approach to discrete simulation. It argues that such an object based, three-phase system is just as convenient as a process based approach.

1 SIMULATION AND OBJECT ORIENTATION

1.1 Origins

Recent years have seen a substantial growth of interest in the subject of object orientation. Though hardly a novelty in the wider world of computing, actual applications in the computer simulation arena are still somewhat sparse. Commercial discrete simulation systems such as MODSIM II (from CACI) are available and embody many of the concepts of object orientation. Going back into the 1960s, it seems widely accepted that the first system to embody most features of object orientation was SIMULA (Dahl & Nygaard, 1966). Thus discrete simulation has been intimately linked with object orientation since the inception of the latter.

Recent years have seen a number of attempts to produce discrete simulation systems using a variety of object oriented approaches and programming languages. Examples include systems written in SmallTalk (for example, Ulgen, 1986 and Knapp 1986 & 1987) and C++ (for example, Blair and Selvaraj, 1989). 1986 saw the publication *System Simulation, Programming Styles & Languages* (Kreutzer, 1986) which attempted to show how object oriented approaches to simulation could be taken in procedural languages such as ISO Pascal, as well as using SmallTalk. There have also been conferences devoted to cross-fertilisation between simulation and object orientation as reported in various conference proceedings.

1.2 Process Interaction

SIMULA implemented an approach which is now known as process-interaction and which is well understood in the discrete simulation community. In this approach, each dynamic object within a model is given one or more processes through which it moves during its life within the system. A process becomes a sequential list of the actions in which the object (or entity) may engage as the simulation proceeds. As simulation entities are created in such an approach, they are seen as members of particular classes and are destined to operate within the processes defined for their class. Thus each entity is regarded as an instance of a class and inherits a template which defines its life within the system. The task of the simulation system's executive is to track each entity through its process template, making sure that the entity operates to plan. This is the essence of the process description approach (Davies & O'Keefe, 1989).

SIMULA, being based on Algol had access to pseudo-concurrency which permits the operation of a routine to be suspended whilst it calls another routine or process into operation. Thus a calling process may interact with a called process, leading to the notion of process-interaction in which the processes of classes of entities interact with one another to lead to the characteristic behaviour of the simulation model. Process interaction, then, goes beyond the basic notion of process description in that processes are assumed to interact with one another. This then allows the entities to be simulated as if they interacted amongst themselves.

1.3 Object Orientation

Given that the processes belong to the entity classes, each created instance of the entity class can be permitted to engage in the same sequence of actions of its process. Thus the actions could be conceived as of operations which, in some sense belong to the entity class in question.

In all discrete simulations, each entity is represented by some form of data structure which must, at the very least represent the current state of the entity and any known future states. Thus an entity class can be represented by a set of operations (the process) which operates on a set of internal state variables which can be hidden from other entity classes in the simulation model. Thus, if the description of a class encompasses the data which represents the entity and also the action sequence, or process, in which the any entity may engage if it is a member of this class.

This of course is the idea of *encapsulation* which is, nowadays, associated with object orientation. Using the conventional definitions of object orientation, encapsulation occurs when class definitions include not just state data, but also a complete list of the operations in which the class member may engage. Thus, in a language such as C++, the class definition includes data members and function members. The data members being used to define the internal data structures of a class and the function members define what operations can be performed on the data members. Each encapsulation presents an interface with other classes, but the internal data and function members need not be visible to other classes. Viewed at its simplest, therefore, encapsulation provides data hiding via modularisation.

But encapsulation is not object orientation, for an OOPS approach requires (Wegner, 1990) that some abstraction mechanism be provided. If descendant classes can be defined so that they inherit all or some of the features of their ancestor classes, then this brings the second feature of object orientation - *inheritance*. This is where the links with discrete simulation terminology appear at their strongest, for inheritance provides the possibility of defining one entity class in terms of another. Thus the simulation objects, or entities, can be successively refined as a simulation program is developed in a parsimonious way. It also permits the provision of proper simulation libraries which can, in theory at least, be enhanced without the need for access to source code.

The final element of object orientation is *polymorphism*, which is essential if a program is to include dynamic objects whose type (class membership) may vary. Polymorphism is the ability to overload the meaning of an operator, function or procedure call. In this way, as a descendant class inherits the methods of its ancestors, these methods may need to be re-defined to cope with special features of the descendants. Thus one method call may have different meanings to different members of a class hierarchy, they objects responding to the message in different ways depending on their class membership.

Within a simulation model, all of these features are valuable. Encapsulation, because it permits a system provider to package together system data and operations in a way which is safe and which allows important data to be hidden from the system user. Inheritance permits objects to be defined in terms of other objects, thus avoiding the need to go on re-inventing the wheel. The addition of polymorphism permits code re-use, as function calls and operators can be overloaded to meet the needs of newly defined descendant classes.

Thus the attraction of object orientation to discrete simulation is clear. But need such an approach be linked solely to the use of a process-interaction approach, given that such approaches have been severely criticised, in the UK at least?

2 THREE PHASE SIMULATION

2.1 A Brief Description

This approach, popular in the UK, originated in manufacturing simulations in the Steel industry and was first described by Tocher (1963) and is covered in considerable detail in Pidd (1992). As with all approaches to discrete simulation, it requires the analyst to atomise the operations of the system being modelled. For a process-interaction model, the atoms are the processes, which are regarded as sequences of operations which can be interrupted as other processes are called at re-activation points. A three-phase approaches requires a rather finer atomisation in which each atom (known confusingly as an activity) is concerned with only the immediate consequences of a state change in the system.

Consider a simple queuing system in which the process of a customer can be described as;

Arrive generate time of next arrival
then
Wait until at the head of the queue and the server is free
then
engage the server & generate service time
Until the service is over
then
release the server.

Where the underlined code indicates re-activation points.

For a three-phase approach, this process would be further atomised into two kinds of operation. Thus which follow only from the passage of time (known as Bs) and those which depend on other conditions within the model (known as Cs). Hence the process is represented by 2 Bs and a C as follows.

Bs: *Arrive* (Add customer to the queue, generate time of next arrival)
End serve (Release server)
C: *Begin serve* (If at head of queue and If server free, *then* engage the server and generate the service time)

It should be clear, that for simple processes, the two process interaction and three-phase approaches amount to pretty much the same thing. This must, of course, be the case for one or both models would be invalid if their logic produced different results. The difference, at this level, lies in the degree of atomisation.

2.2 Three-phase and process interaction

There are two major differences between the process-interaction approach and the three-phase approach. The first is that the three-phase approach has no need to pretend that each operation must be the prime concern of a single entity class. This is particularly important for the Cs, for such conditional activity may depend on the states of a variety of entities in different classes and, most importantly, may cause the states of a number of entities in different classes to change at the same time.

The operation of the Cs is controlled by the simulation executive which checks that the necessary logical tests are passed so as to permit the Cs to begin. Thus, as many entity classes as necessary can be included in the test-head of a C, which need not, therefore, belong to any particular entity class. In a process-interaction approach, each process belongs to a particular class and thus each activity, as a process segment, belongs to a class. With complicated multi-class activity it is by no means clear that this is the best approach.

The second difference is that the code for an activity (whether B or C) may commit any entity to another B sometime in the future. That is, the control of the entity is explicitly handed over to the executive for that intervening period. Thus a B is scheduled pretty much as an event might be in an event-based simulator. In a process-interaction based model, this handover of control is implicit as the entity is delayed in its process or may have to activate other processes. To do the latter properly requires co-routines on single processor computer systems.

In a process-based approach it is clear that the equivalent of Bs can belong to entity classes and each B equivalent can thus be encapsulated with an entity class. However, it is by no means clear what should be done about the equivalent of the Cs which occur, in process-interaction mode, at conditional re-activation points.

3 OBJECT ORIENTATION AND THREE-PHASE SIMULATION

The preceding discussion suggests how object orientation and three-phase simulation might be brought together. Many of the features will be the same as in a process-interaction implementation, the major difference being the ways in which the set of Cs (Conditional activities) need to be treated. Thus a possible object hierarchy presents itself as suitable for a three-phase simulation system and this is illustrated in Table 1.

Table 1: Simple Object Hierarchy for Three Phase Simulation

	DESCENDANT CLASSES	
BASE LEVEL	FIRST LEVEL	SECOND & SUBSEQUENT LEVELS
BaseType	Ring	Queue
BaseType	GEntity	Specific entities for a model
BaseType	Resource	Specific resources for a model
BaseType	Calendar	Executive
BaseType	Rand	Random sampling objects
BaseType	Tally	Data collection objects
BaseType	C activities	

Where the following definitions apply.

BaseType:	a simple abstract class from which others descend.
Ring:	some way of maintaining lists of BaseTypes and descendants. Rings of rings are possible.
Queue:	ordered sets of BaseTypes and descendants.
GEntity:	a general entity template which maintains state information for all entity classes.
Resource:	countable static resources used by entities.
Calendar:	a chronological list (event list).
Executive:	the control program.
Cs:	the Cs, organised into a list via a Ring.

Thus for a particular model, entity classes descend from the GEntity class and resources from the Resource class. The public member functions of entity classes include the Bs. The Cs are placed in a CList which is a data member of the Executive object. Objects to carry out random sampling and run-time collection and display can obviously also be added as descendants of the BaseType. Thus a GEntity could be made to inherit from these extra objects as well.

3.1 Implementations

The system described above has been implemented in Turbo Pascal v6 and also in Turbo C++ v2. The availability of multiple inheritance in C++ made the task rather simpler and produced fewer anomalies. The addition of object oriented features to languages such as C and Pascal can be viewed as an attempt to make these procedural programming languages closer to a declarative style. This, if successful, should make discrete simulations easier to write as the user can be less concerned about the implementation of libraries provided for their use than has to be the case in C or Pascal.

3.2 The C++ Implementation

This object hierarchy has been implemented in C++, using the cheap Borland Turbo compiler, without any significant problems. The simulation objects themselves (descendants of GEntity) are, of course, dynamic and therefore recourse had to be made to the usual dangerous C++ type casting in order to cope with these objects. This particular implementation is divided between several source and header files as follows.

GenLib:	containing a library of general purpose C++ functions for smoothing I/O and screen display. These need not be written in an object oriented style.
OOPGen:	containing the definitions and implementations of the non-simulation specific C++ functions and variables which create the basic classes of Table 1.
SimObs:	containing the simulation specific objects.
Rnd:	containing sampling routines and data collection facilities.

These files are then used by the final simulation program itself, this takes the class definitions and creates, where necessary, descendant classes to fit the application.

3.3 An example of its use

The C++ implementation is forming the basis of the simulation component of a Geographic Decision Support System (GDSS) for use by emergency planners in the UK. This GDSS is to help emergency planners in planning evacuation from natural or man-made disasters in the UK and is, as an initial case, focused on nuclear power plants close to urban centres. The GDSS is built on a database of digitised maps which use the ARC/INFO systems. The simulation element aims to show the effect of different evacuation policies in the event of a major crisis. Hence the simulator has to show vehicles and other objects moving through a road network in accordance with routing rules and algorithms.

The core of this can be simulated rather simply in three-phase mode using the C++ system described earlier. The following classes need to be defined from those in the basic system.

Network:	this is a descendant of Ring and contains the currently feasible routes in the road network. A route consists of a sequence of *Locations*.
Location:	this is a descendant of base type and is, in effect, a short segment of road which a vehicle must traverse.
Vehicle:	a descendant of GEntity and, thus, the main dynamic simulation object. Sub-classes of vehicle are formed by defining new descendant classes. Each vehicle has, amongst its data members, its current location and its currently feasible network.

Thus the simulation executive moves the vehicles through the network using the locations and assumed speeds. Congestion is represented by vehicles attempting to occupy the same location. The current state of evacuation can be shown on-screen by instructing the vehicles to reveal their locations.

4 SOME CONCLUSIONS

A three-phase approach to discrete simulation is easily implemented in C++ using an object oriented approach. This differs from a process based approach in that conditional activity is not linked to specific entity classes but is modelled as an attribute of the total simulation system. If necessary, these Cs can be defined as part of the simulation executive. Though such a system can be programmed in a single inheritance mode, such as with Turbo Pascal 5.5 and later, multiple inheritance makes the task much easier.

Use of this object library in the evacuation simulator should indicate whether it makes program development easier, faster and safer.

REFERENCES

Blair E.L. & Selvaraj S. (1989) *DISC++: a C++ based library for object oriented simulation.* Proc 1989 Winter Sim Conf, pp301-307.

Dahl O. & Nygaard K. (1966) *SIMULA - an Algol based simulation language.* Comm ACM, 9, 9, pp671-678

Davies R. & O'Keefe R.W. (1989) *Simulation in Pascal.* Prentice-Hall, Englewood Cliffs, NJ.

Knapp V. (1986) *The SmallTalk simulation environment.* Proc 1986 Winter Sim Conf, pp125-128.

Knapp V. (1987) *The SmallTalk simulation environment.* Proc 1987 Winter Sim Conf, pp146-151.

Kreutzer W. (1986) *System simulation, programming languages and styles.* Addison-Wesley Co, Sydney.

Pidd M. (1992) *Computer Simulation in management science, third edition.* John Wiley & Sons Ltd, Chichester.

Tocher K.D. (1963) *The art of simulation.* English Universities Press, London.

Ulgen O.M. (1986) *Simulation modularity in an object oriented environment using SmallTalk-80.* Proc Winter Sim Conf, pp474-484

Wegner P. (1990) *Concepts and paradigms of object oriented programming.* OOPS Messenger 1, 1, pp7-83, ACM Press

AUTHOR BIOGRAPHY

MIKE PIDD is a professor in the Management School of Lancaster University, UK. His research interests focus on the creation of useable computer-based models, especially in manufacturing. He is well known for two books on computer simulation methods, *Computer Simulation in Management Science (3rd ed)* and *Computer Modelling for Discrete Simulation*, both published by John Wiley. He was awarded the President's Medal of the Operational Research Society for a paper on computer simulation.