

Overview of Modeling and Simulation and Object  
Oriented Software Design Pertinent to  
Development of a Hierarchical Simulation  
Platform

Peter Steadman

January 27, 2003

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Overview of Simulation Techniques</b>	<b>2</b>
2.1	Simulation modeling techniques in general . . . . .	2
2.1.1	Static, stochastic simulation models . . . . .	3
2.1.2	Discrete-event simulation models . . . . .	4
2.2	Discrete-event simulation theories . . . . .	5
2.3	Hierarchical simulation . . . . .	7
2.3.1	DEVS and DEVS-Scheme . . . . .	7
2.3.2	Swarm . . . . .	8
2.3.3	Others . . . . .	9
<b>3</b>	<b>Select catalog of possibly pertinent Design Patterns</b>	<b>10</b>
3.1	Observer . . . . .	10
3.1.1	Description . . . . .	11
3.1.2	Possible Applicability . . . . .	11
3.2	Event Channel . . . . .	11
3.2.1	Description . . . . .	12
3.2.2	Possible Applicability . . . . .	13
3.3	Composite . . . . .	14
3.3.1	Description . . . . .	14
3.3.2	Possible Applicability . . . . .	14
3.4	Singleton . . . . .	14
3.4.1	Description . . . . .	14
3.4.2	Possible Applicability . . . . .	14
3.5	Facade . . . . .	14
3.5.1	Description . . . . .	15
3.5.2	Possible Applicability . . . . .	15
3.6	Memento . . . . .	16
3.6.1	Description . . . . .	16
3.6.2	Possible Applicability . . . . .	17

## 1 Introduction

Our goal is to develop a software platform to support the implementation of what we are terming *Hierarchical Simulations*. Our expertise, and therefore our focus and vision is in the field of Systems Biology, so the platform is primarily intended to support simulation projects in this field, however we don't intend to expressly limit the platform to this domain. By Hierarchical Simulation we mean compound simulations in which the elements of which the whole is compounded are models of phenomena that occur at the various scales that are of interest in biology. A Hierarchical Simulation might, for instance, be comprised of models of molecules that are hierarchically "nested" into models of organelles that are "nested" into models of cells and so on up, perhaps, to the level of an ecosystem. Each of these levels represents a separate field of expertise, and one of the advantages of a hierarchical approach is that it can provide a framework in which these fields can be encapsulated, and the different degrees of understanding available at the various scales can be incorporated without regard to the level of detail incorporated at other scale levels. This encapsulation of expertise is one of the major goals of the platform we propose.

This approach (the hierarchical approach to simulation) is not revolutionary (or even original), and in the first part of this paper we provide some overview of the history of this approach and, to a small degree, of computer modeling and simulation in general. One of the goals of this paper is to inform ourselves of previous work in this domain so that we can profit from it and build upon it in our work. This literature is vast, however, and we don't attempt to cover it comprehensively.

Much of the literature on Modeling and Simulation concerns the problem of parallelization. Since (at least at the outset) parallel processing is beyond the scope of our development effort, we ignore this rich literature for the purposes of this treatment.

For the design and implementation of the platform we (along with most modern simulation practitioners) embrace Object Oriented technology, and the second part of this paper provides a small catalog of software Design Patterns within the Object Oriented paradigm that we feel might be useful in the platform's design. We have perused the cannon of Design Patterns (this is a very active field in the Software Engineering discipline) and selected those that we feel will most likely be applicable. The purpose of this part of the paper is to inform and focus our design effort.

## 2 Overview of Simulation Techniques

### 2.1 Simulation modeling techniques in general

Averill Law and David Kelton [9] describe a categorization of simulation techniques along three axes:

1. static vs. dynamic simulation models,

2. deterministic vs. stochastic simulation models, and
3. continuous vs. discrete simulation models.

Static models are either models of time-independent systems, or models of a system at a particular time.<sup>1</sup> Dynamic models, of course, are those that model systems as they evolve over time.

Models that rely on the generation of random variables in deciding how to change state are stochastic, those that don't are deterministic. A deterministic model's trajectory is determined when the inputs to the model are given. A stochastic model's trajectory is a random variant. Differential equations based models, for instance, are deterministic whereas most other modeling techniques involve some kind of probabilistic component.

Discrete simulation models are ones in which the model "jumps" from one state to another at discrete time points. The system being modeled might be one in which this aspect is natural, or it might be a naturally continuous system that is being approximated by a discrete model.<sup>2</sup> Differential equations models are the most common examples of continuous simulation models.

### 2.1.1 Static, stochastic simulation models

To begin with, *Monte Carlo* simulation is defined by Law and Kelton [9] and by Ross [12] in a very precise way that restricts it to modeling static, deterministic systems<sup>3</sup>. Law and Kelton [9] mention that other authors "define Monte Carlo simulation to be *any* simulation involving the use of random numbers," however, according to their definition and Ross', Monte Carlo simulation is specifically a technique for estimating solutions of definite integrals (or systems of related definite integrals) whose integrand(s) is(are) not analytically integrable. It is a rather brute force approach (comparable to some numerical approaches) in which the integrand is evaluated a large number of times at random points that are uniformly distributed along the interval over which the integral is being evaluated.

The following explication is due to Ross<sup>4</sup>. Suppose that we want to evaluate the integral,

$$I = \int_0^1 g(x)dx$$

---

<sup>1</sup>It would seem that static systems would not be of great interest in the context of simulation, however Law and Kelton as well as Sheldon Ross [12] give a very restrictive definition of a simulation technique called *Monte Carlo* simulation which is applicable only to such static systems. More on this later.

<sup>2</sup>An example of a "naturally" discrete system is a bus in which passengers board and exit only when the bus arrives at a bus stop[6].

<sup>3</sup>"We define *Monte Carlo* simulation to be a scheme employing random numbers ... which is used for solving certain stochastic or deterministic problems where the passage of time plays no substantive role. Thus Monte Carlo simulations are generally static rather than dynamic." [9] pg. 113.

<sup>4</sup>[12] pgs. 38-40 with some embellishment taken from his section 2.1, *Elements of Probability*.

We note that because the expectation of a continuous random variable  $X$  having the probability density function  $f(x)$  is defined to be

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

and that since, for a uniform distribution  $U$ ,  $f(u) = 1$ , we can replace  $x$  with a uniformly distributed random variable on the interval  $(0, 1)$  and express  $I$  as

$$I = E[g(U)]$$

Changing to a discretization of the problem, so that it can be easily attacked computationally, we can now sample  $U$  in a manner that produces  $U_1, \dots, U_k$  which are independent, uniform random variables on the the interval  $(0, 1)$ . It follows that the the random variables  $g(U_1), \dots, g(U_k)$  are also independent and also uniformly distributed. The discrete version of the definition of the expectation is

$$E[X] = \sum_i x_i P\{X = x_i\}$$

and in the case of a uniform distribution of a discrete random variable  $U \in \{u_1, \dots, u_k\}$ ,  $P\{U = u_i\} = 1/k$ . It follows that

$$\sum_{i=1}^k \frac{g(U_i)}{k} = E[g(U)]$$

in the discrete case. As  $k \rightarrow \infty$  the discrete case approaches the continuous case in which  $E[g(U)] = I$ . So, by generating a large number of random numbers  $u_i$ , we can produce an approximation for  $I$ .<sup>5</sup> The more random points evaluated, the better the approximation.

Because the integrals are definite, there is no variation in the output of the simulation even though a series of random variates have been used to compute the solution. This would make it seem of limited interest when simulating biological or ecological systems, however we note that the static nature of Monte Carlo simulation does not preclude it from being used in the simulation of a time varying system if, for instance, the integrand has time as a parameter.

### 2.1.2 Discrete-event simulation models

The largest part of the computer modeling and simulation literature regards discrete, dynamic, stochastic models, which therefore fall into the category of *discrete-event simulation models*[9]. The discrete time points that are modeled in discrete-event models are those at which the system changes state. Some sources (e.g. [18]) draw a distinction between discrete-event models and *discrete-time* models. *Discrete-time models* are a sub category of discrete-event models in

<sup>5</sup>Intervals other than  $(0, 1)$  can be evaluated by using substitutions to formulate another function  $h(y)$  where  $y$  is a function of  $x$  and the boundaries of the interval such that  $I = \int_0^1 h(y)dy$ .

which all time steps are considered for all elements of the model. Discrete-event models that are not discrete-time models consider only those time steps at which state changes occur. That is, an “event-based approach jumps from one interesting event to the next, omitting the uninteresting behavior in between,” [18] (pg. 67). As an alternative, Law and Kelton [9] describe, “two principal approaches [that] have been suggested for advancing the simulation clock: *next-event time advance* and *fixed-increment time advance*.” (pg. 8). They go on to say that, “the first approach is used by all major simulation languages and by most people coding their model in a general-purpose language.”

One interesting discrete-event modeling scheme (on which there is a considerable literature) is the method of *cellular automata*. A cellular automata model is one in which both time and space are discretized. It is comprised of an array (one, two, or multidimensional) of “cells” each of which is governed by an identical set of rules sometimes referred to as its “program”. Each cell has a separate “state” which is one of a discrete, finite set (each cell is a “Finite State Machine”), and has a “neighborhood” of other cells (usually those that are proximal) whose states influence its state. All cells change their state together, according to their rules, at discrete times. Cellular automata were introduced by John von Neumann and Stanislaw Ulam [2] in the late ’40s, and have a very well developed theory. See for instance Wolfram [15].

Also worth mentioning in the realm of discrete-event model paradigms are *Markov chain Monte Carlo methods*. A Markov chain Monte Carlo method is a method for approximating the distribution of a random vector of random variables whose elements are dependent on one another. A Markov chain is “a collection of random variables  $X_t$  (where the index  $t$  runs through 0, 1, ...) having the property that, given the present, the future is conditionally independent of the past.” [10] That is to say that the value of the vector at time  $t$  is independent of the values at  $t < t - 1$  but dependent on the values at  $t = t - 1$ . If you know the mass function (or know it up to a constant multiplier), of the vector you are interested in simulating, certain types of Markov chains (namely those that are *irreducible* and *aperiodic*) can be used to approximate the distribution. When these chains are generated stochastically, the technique is called Markov Chain Monte Carlo. For a formal and thorough account, see Ross [12] pgs. 223–250.

## 2.2 Discrete-event simulation theories

Discrete-event simulation in general has a number of very well developed theories and formalisms. For instance, Bernard Zeigler [17], [16], [18] in the early 70’s introduced *Discrete Event System Specification* (DEVS). DEVS is, according to Zeigler, a discrete event system *formalism*. According to Zeigler, a DEVS,

“...is a structure

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

$X$  is the set of input values

$S$  is a set of states

$Y$  is the set of output values

$\delta_{int} : S \rightarrow S$  is the *internal transition* function

$\delta_{ext} : Q \times X \rightarrow S$  is the *external transition* function, where

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$  is the *total state* set

$e$  is the *time elapsed* since last transition

$\lambda : S \rightarrow Y$  is the output function

$ta : S \rightarrow R_{0,\infty}^+$  is the set positive [sic.] reals with 0 and  $\infty$ <sup>6</sup>

This is all extremely formal and, without in depth treatment, quite opaque. However, it is of salient interest to us because DEVS can be “coupled” and composed into compound DEVS<sup>7</sup>, and he uses this property to create modular model hierarchy trees that seem quite consonant with the hierarchical, modular system we envision. We’ll treat this in a bit more depth in the next section.

Law and Kelton[9] also develop a formal theory of discrete-event simulation. Their theory is developed differently from Zeigler’s (which, as I’ve said, he refers to as a “formalism”). Law and Kelton seem to have approached the development of their theory from an observational science perspective, analyzing the properties of an existing field. Zeigler’s more *ab initio* approach was initiated in the nascent years of computer simulation science. It is interesting to note that most of the components in Law and Kelton’s analysis appear in some guise in the analysis and design work we have done so far in development of our Hierarchical Simulation Platform. According to Law and Kelton’s theory, most simulation models using next-event time advance include the following components:

**“System state:** The collection of state variables necessary to describe the system at a particular time.

**Simulation clock:** A variable giving the current value of simulated time.

**Event list:** A list containing the next time when each type of event will occur.

**Statistical counters:** Variables used for storing statistical information about system performance.

**Initialization routine:** A subprogram to initialize the simulation model at time zero.

**Timing routine:** A subprogram that determines the next event from the event list and then advances the simulation clock to the time when that event is to occur.

---

<sup>6</sup>[18] pgs. 75–76.

<sup>7</sup>He describes this property as *closure under coupling*[16].

**Event routine:** A subprogram that updates the system state when a particular type of event occurs (there is one event routine for each event type).

**Library routines:** A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model.

**Report generator:** A subprogram that computes estimates (from the statistical counters) of the desired measures of performance and produces a report when the simulation ends.

**Main program:** A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.”<sup>8</sup>

It should be noted that both Law and Kelton[9] and Zeigler[18] describe discrete event simulation as maintaining a list of events sorted by their scheduled event times, and executing events sequentially on that list. We mention this just because it differs from the scheme we have discussed in which events are triggered by internal state of the component models. An “Event List” does not seem the most natural way to model dynamic systems. Though this may quite often be an appropriate simplification, we expect that quite often it is not. It seems to introduce a notion of pre-determination that is at odds with the notion of a system’s evolution through time. Life does not have an event schedule. (At least, not so far as we know.)

## 2.3 Hierarchical simulation

### 2.3.1 DEVS and DEVS-Scheme

Hierarchical simulation has been discussed in the literature at least since the ’80s. We mentioned above that Zeigler[16] described a “hierarchical, modular specification of discrete-event models,” and an implementation of an environment (DEVS-Scheme<sup>9</sup>) for implementing these models and running simulations. One of the main goals of DEVS-Scheme was to promote model reuse, and one of the major concepts in DEVS is the *model base*. DEVS models have inputs and outputs which are called “ports”. Two models are “coupled” to form compound models by attaching the output ports of one to the input ports of the other. We describe this design here because it is not the only possible design for composing models hierarchically, and it is something we should consider when designing our system. The design is described diagrammatically in figure 1.

---

<sup>8</sup>[9] pgs. 10–11.

<sup>9</sup>It was written in Scheme. Scheme, a dialect of LISP, is one of the earliest object-oriented languages. Object-oriented programming, LISP, Scheme, and simulation all grew up together in the AI community.

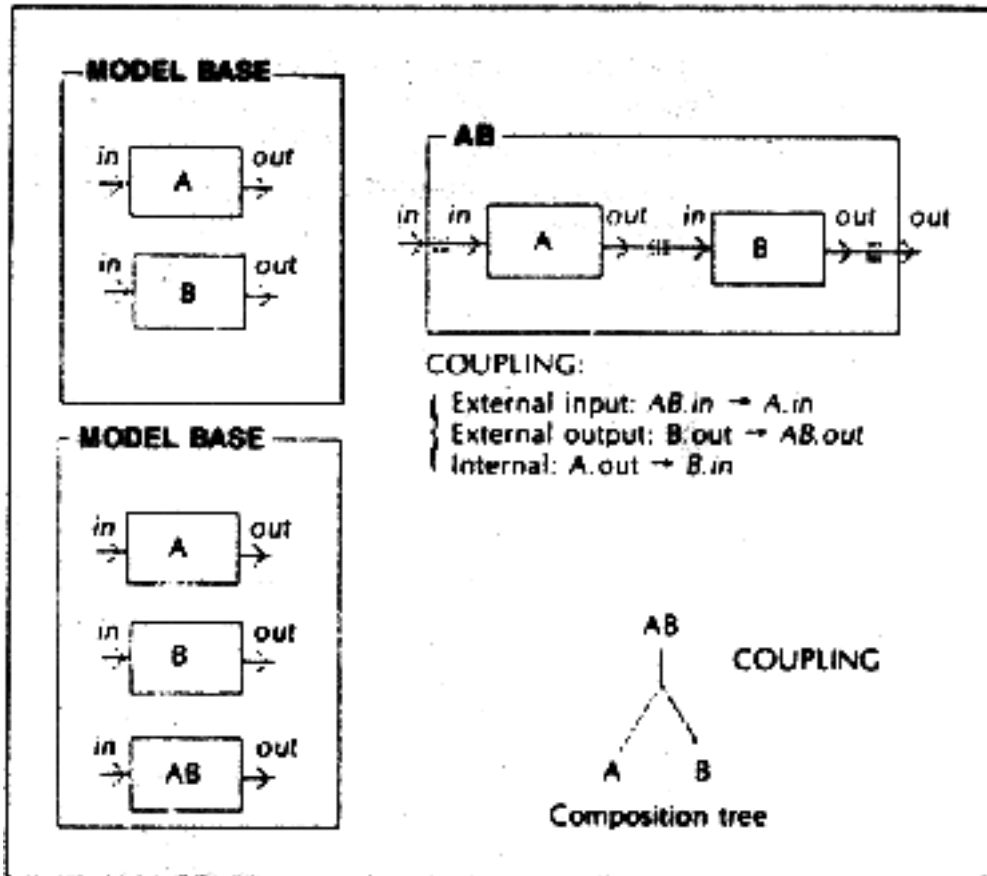


Figure 1. Model base concepts.

Figure 1: Figure from [16] depicting the design of a DEVS model base, and the mode in which models are coupled to form compound models.

### 2.3.2 Swarm

A system that overlaps quite strikingly with the ideas we are developing is the *Swarm Simulation System*. Swarm is a software package for multi-agent simulation of complex systems, originally developed at the Santa Fe Institute[13]. In [4], Marcus Daniels describes it this way,

Swarm is a set of libraries that facilitate implementation of agent-based models. Swarm's inspiration comes from the field of Artificial Life. Artificial Life is an approach to studying biological systems that attempts to infer mechanism from biological phenomena, using the elaboration, refinement, and generalization of these mechanisms

to identify unifying dynamical properties of biological systems.

Swarm adopts a modeling formalism in which collections of independent agents interact via discrete events. Such a collection of agents, along with a schedule of events, is termed a “swarm.”

Swarm has the capacity for implementing hierarchical models. In [11], Minar et. al. describe building models at various levels this way:

In addition to being containers for agents, swarms can themselves be agents. A typical agent is modeled as a set of rules, responses to stimuli. But an agent can also itself be a swarm: a collection of objects and a schedule of actions. In this case, the agent’s behavior is defined by the emergent phenomena of the agents inside its swarm. Hierarchical models can be built by nesting multiple swarms. For example, one could build a model of a pond inhabited by single celled animals. At the highest level, a swarm is created that contains agents: the swarm represents the pond, and each agent represents one animal. The behavior of cells could be defined simply as some algorithm, but a cell is itself a collection of organelles: a nucleus, mitochondria, endoplasmic reticulum. Another way to represent a cell is as a swarm of agents, the organelles. Two models are being combined: the pond as a swarm of cells and the cell as a swarm of organelles.

Figure 2 depicts such a hierarchical modeling approach implemented in Swarm.

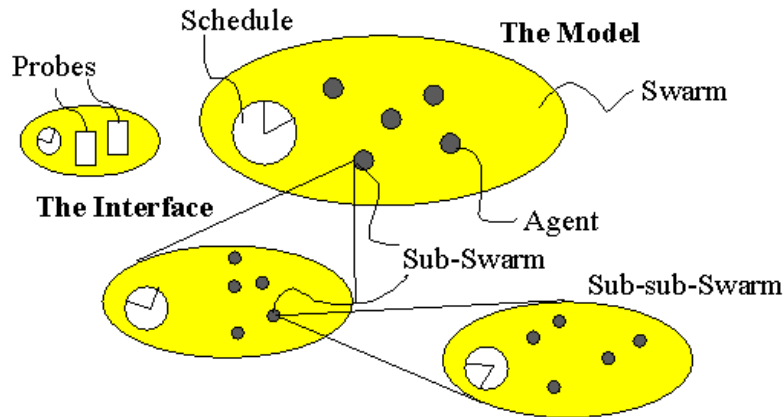
Swarm’s development began in 1994, and the system continued to be actively developed at least through 2000. It’s not clear how much development continues on the system, but there is an active user group. SwarmFest 2003 was recently announced on the swarm website [13]. Swarm has been used to build simulations in a wide variety of domains. For example a book was published in 2001 called *Agent-Based Methods in Economics and Finance: Simulations in Swarm*[7].

### 2.3.3 Others

We believe that there are a number of systems similar (in the sense that they compose models hierarchically) to the two we describe here. For example, in the context of modeling the dynamics of polymers, Doros N. Theodorou describes hierarchical modeling much in the way we mean:

Recent years have witnessed a growing realization, within the materials modelling community, that theoretical and simulation approaches at different time and length scales can be connected together into hierarchical schemes. [...] A modeling hierarchy consists of several levels, each utilizing a model whose parameters are derived from lower (more fundamental, smaller length- and time scale) levels and providing input to higher (more coarse grained, larger length- and time scale) levels.[14]

## The recursive structure of Swarms



4/1/99

Benedikt Steffansson &lt;benedikt@ucla.edu&gt; - March 1999

22

Figure 2: Figure from [8] showing how hierarchical models can be developed in Swarm.

### 3 Select catalog of possibly pertinent Design Patterns

Design patterns in software architecture and object-oriented design are abstractions of solutions to commonly occurring software design problems. The notion of “patterns” in software design was adopted from architecture. Christopher Alexander coined the term “pattern language” in his books *A Timeless Way of Building* and *A Pattern Language* which provided a catalog of solutions to commonly occurring design problems in architecture. The first catalog of patterns in object-oriented software design was Gamma et. al. [5] which came out in 1995. There is now a fairly substantial literature. Here we have selected a few that we felt would be likely to play some role in our development. We describe them briefly here only to collect their descriptions in one place so that they can more easily influence our design discussions.

#### 3.1 Observer

Define an observer-observed relationship between objects such that the observers of an object are immediately notified of state changes in the observed. This

pattern was introduced in Gamma et. al.[5], it is also described in Buschmann et. al. [3].

### 3.1.1 Description

This is a pattern in which an object broadcasts changes in its state to other objects that have registered to “listen for” those changes. It is also known as “publish and subscribe.” Objects subscribe to events from some object, and that object publishes to its subscribers. It is used to define many-to-one dependencies so that when one object changes state, all its dependents are notified and can be updated automatically.

### 3.1.2 Possible Applicability

This could be an elegant way of implementing a system for generating reports on data evolving in subparts of a simulation. Objects within the hierarchical levels could have “observables” for which the report generating subsystem could register. This seems very much in line with the “probe” approach adopted by Swarm.

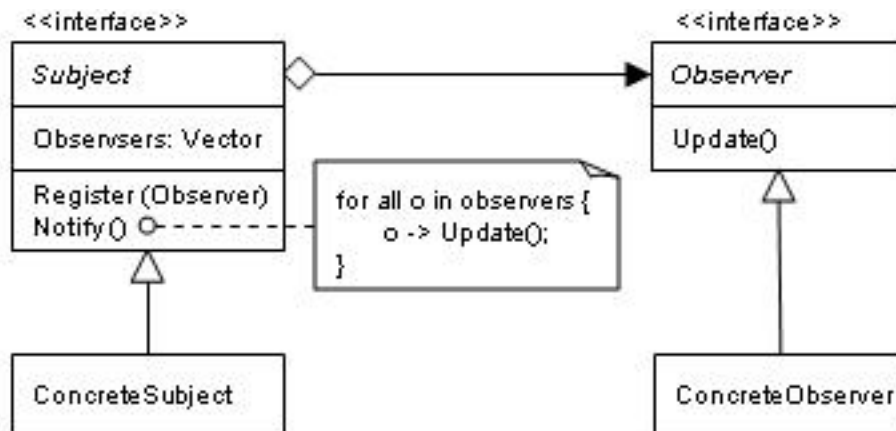


Figure 3: UML class diagram showing the class structure of the Observer pattern.

## 3.2 Event Channel

Allow occurrences in one object to influence other interested objects without introducing dependencies. Buschmann et. al. [3] describe this pattern as a variant of the Observer pattern.

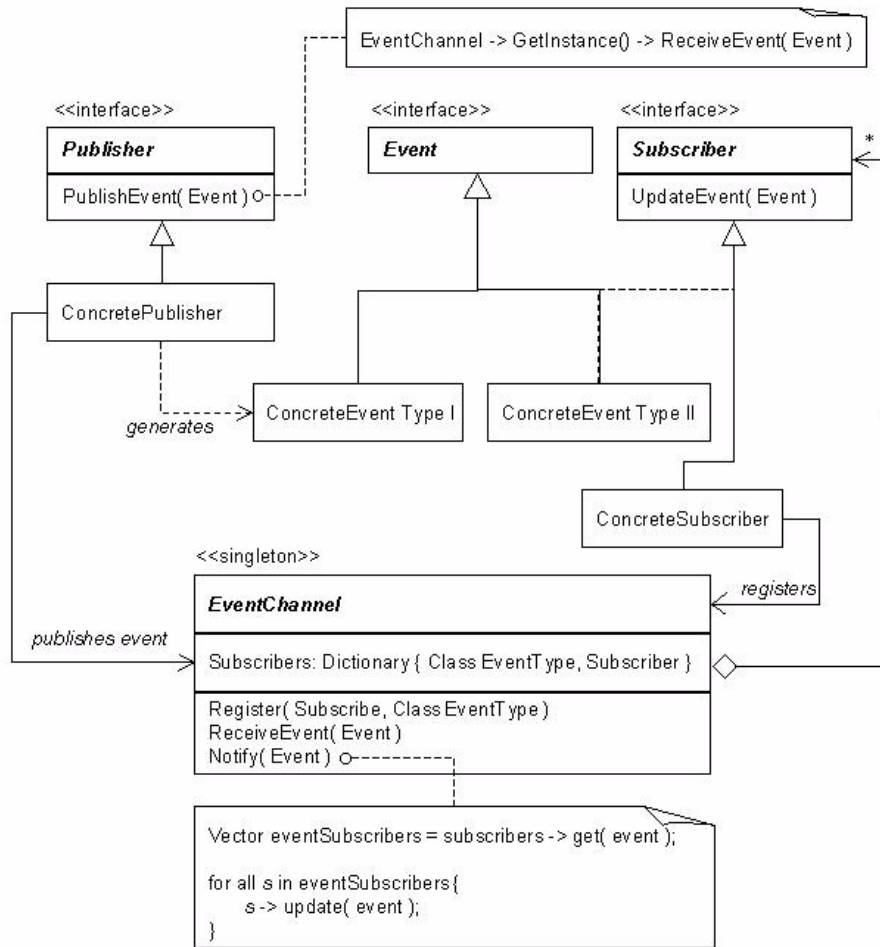


Figure 4: Class diagram showing the class structure of the Event Channel pattern.

### 3.2.1 Description

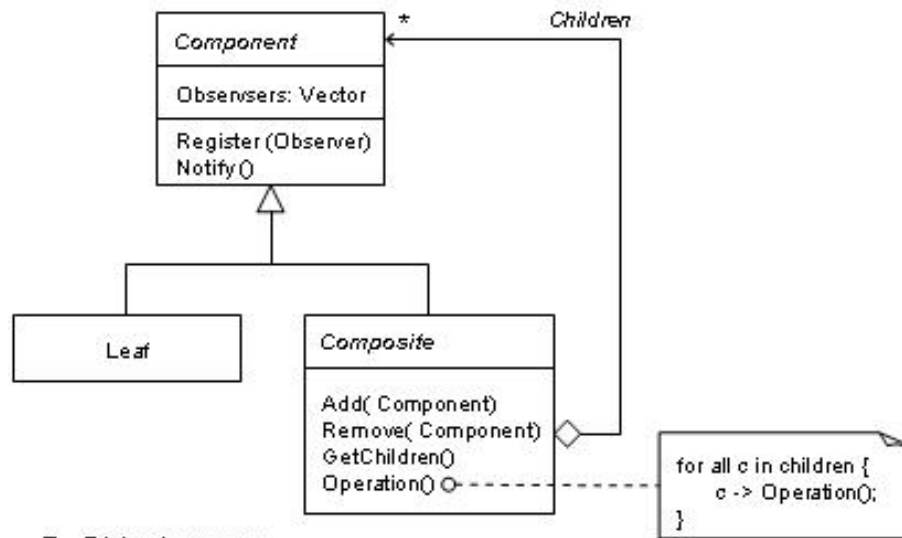
Often referred to as a “Bus,” Event Channel is a variant of the Observer pattern that more strongly decouples the observed and the observer. In this pattern there can be more than one publisher of a certain observable (which in this context is called an “event”), and observers need only know about the event, and not about its broadcaster. In this variant, a third object (the “event channel”) is placed as a mediator between the observer and observed. To the observer the

event channel acts as an observable, and to the observed, it acts as an observer.

### 3.2.2 Possible Applicability

Some variation of this idea could be useful in removing dependencies between objects that represent models within the hierarchical levels. The objects will have to be influenced by occurrences (state changes) in other objects either in the same level or at different levels, yet we will have to control the dependencies. We want strong encapsulation and independence of one level from the others so that, a) levels can developed independently, b) changes in one level do not require changes in others, and c) levels can be easily added and removed.

#### A. Class space



#### B. Object space

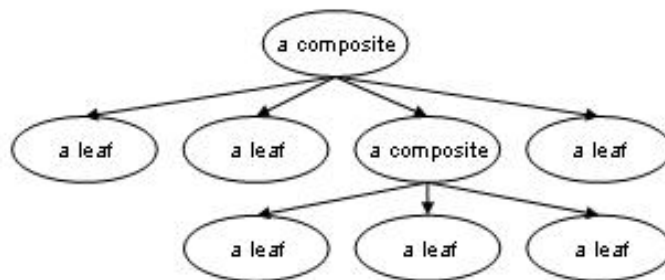


Figure 5: **A** A class diagram illustrating the Composite pattern. **B** A possible object hierarchy constructed using the pattern.

### 3.3 Composite

Compose objects into tree structures to represent part-whole hierarchies. This pattern is described in [5].

#### 3.3.1 Description

In this pattern object hierarchies (as opposed to class hierarchies) that can represent whole-part relationships are constructed. An abstract class is created that can represent either a composition or a part, and the objects representing the parts and objects representing the compositions are both concrete subclasses of this abstract class. The concrete class representing the composition has the difference that it contains methods for adding and removing parts, and for retrieving its children. This pattern is used when you want to be able to ignore the difference between compositions of objects and the atomic objects of which they are composed.

#### 3.3.2 Possible Applicability

It is not clear at this point how we want to construct our hierarchies, and it may be that we find some advantage to constructing them not only in *class-space*, but also (or instead) in *object-space*. The Composite pattern is a means of doing the latter.

### 3.4 Singleton

Ensure that an object have only one instance. This is a basic, simple but powerful pattern described in [5].

#### 3.4.1 Description

Sometimes a class describes something of which there should be only one instantiation. For instance, a system that uses a “clock” class to synchronize objects should probably have only one to ensure that synchronization is actually achieved. The Singleton pattern enforces the singularity of the one object of this class and ensures that there is one universal access point for it.

#### 3.4.2 Possible Applicability

There is likely to be a number of applications for the Singleton pattern in our platform. There might well be one clock, for instance, and one Event Channel with which simulation objects interact.

### 3.5 Facade

Provide unifying interface to a complex subsystem. Also described in [5].

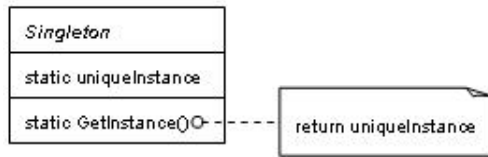


Figure 6: A class diagram illustrating the Singleton pattern.

### 3.5.1 Description

The Facade pattern provides one simple interface to a complex subsystem consisting of multiple entities. An interface is provided for the subsystem as a whole, and it delegates to the various subsystem entities to perform the actions requested of the client. It serves two purposes: it simplifies interaction with the subsystem, and it facilitates decoupling between the subsystem and its clients. By decoupling we mean that responsibilities within the subsystem can change without affecting the clients who still interact only with the Facade.

### 3.5.2 Possible Applicability

The platform subsystem is likely to be somewhat complex. We will likely want to hide its various entities behind a Facade to simplify the job of the simulation application developer. The application developer would have only one entity with which it would have to interact.

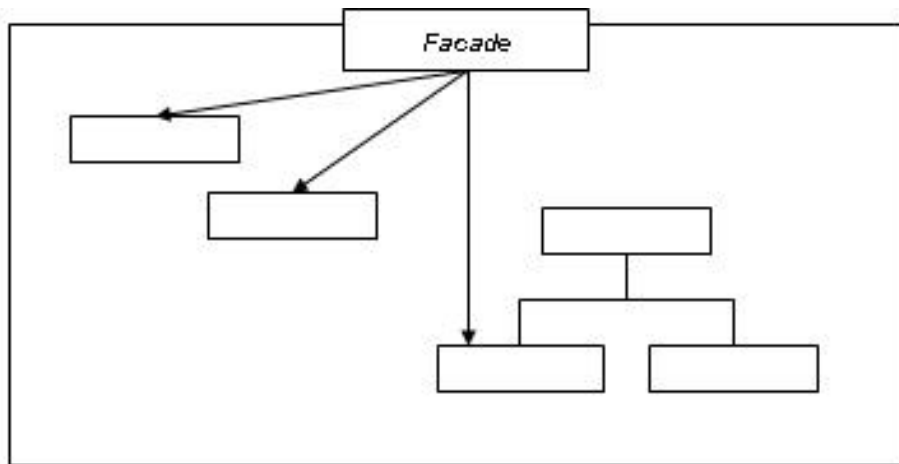


Figure 7: A Facade provides a unifying interface for a complex subsystem.

### 3.6 Memento

Create an ‘image’ of an object so that its state can later be restored. Do so without violating encapsulation. Also described in [5].

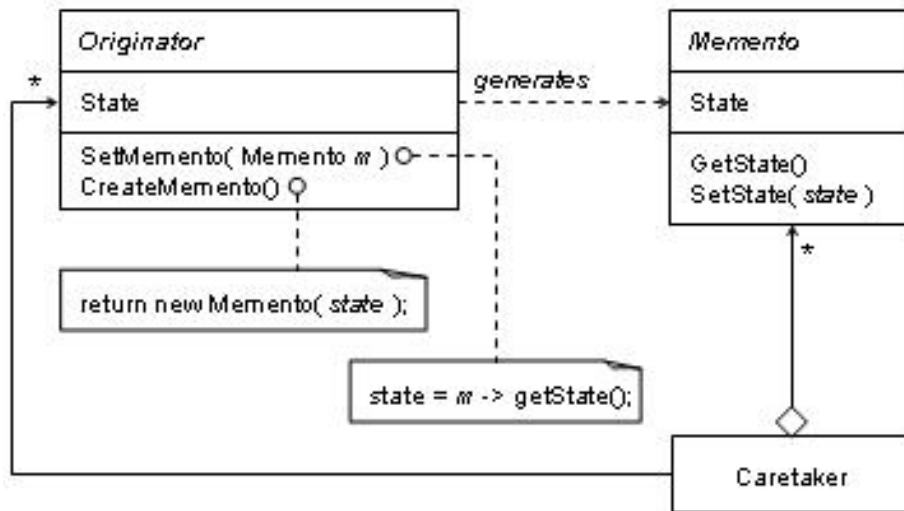


Figure 8: A class diagram illustrating the Memento pattern. The `GetState()` and `SetState()` methods are part of the “wide” interface available only to the `Originator`.

#### 3.6.1 Description

This pattern is typically used for implementing undo mechanisms in applications such as word processors. The Memento pattern provides a way of accomplishing this while maintaining encapsulation. In order to ensure that classes are not overly coupled – that is that they are not overly dependent on each others implementation details – an object’s state information is generally “encapsulated” such that it is inaccessible outside of the object. In the Memento pattern, the object creates a Memento object in which it stores whatever aspects of its current state would be needed to restore the process. The Memento object has two interfaces, one for the originating object, and another for an object, called a CareTaker that manages the mementos. The interface exposed to the originating object is “wide”, displaying all of the saved state. The interface exposed to the CareTaker or anyone else is “narrow” displaying nothing. When the system’s state is being recovered, the CareTaker hands the appropriate memento back to the originating object so that it can restore itself to that state.

### **3.6.2 Possible Applicability**

The platform will have to be able to recover simulations when runs have gone awry or reconstitute a run from a particular time so that the user can provide a different input at that stage and explore a different trajectory. It is likely that serializing the entire process periodically will be too heavy weight, and we will most likely be able to get away with just storing some select state information. A variation of the Memento pattern could provide an elegant solution. In our application, the care taker could periodically collect the required mementos and write them to a file which either it or another object could read when a run is recovered.

## References

- [1] Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, *A Pattern Language*. Oxford University Press, New York, 1977.
- [2] Burks, Arthur W., “Von Neumann’s self-reproducing automata.” In Arthur W. Burks, editor, *Essays on Cellular Automata*, pages 3–64. University of Illinois Press, Urbana, Illinois, 1970.
- [3] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [4] Daniels, Marcus, “Integrating Simulation Technologies With Swarm,” a paper delivered to the “Agent Simulation: Applications, Models and Tools” conference held at University of Chicago in October of 1999. (Available online at <http://www.santafe.edu/mgd/anl/anlchicago.html>.)
- [5] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Welsey, 1995.
- [6] Fishman, George S., *Principles of Discrete Event Simulation*. Wiley, New York, 1978.
- [7] Luna, Francesco, A. Perrone, editors, *Agent-Based Methods in Economics and Finance: Simulations in Swarm*. Kluwer Academic Publishers, Boston, 2001.
- [8] Stefansson, Benedikt, “SwarmFest ’99 Tutorial,” available online at <http://www.swarm.org/intro-tutorial.html>.
- [9] Law, Averill M., and D. W. Kelton, *Simulation Modeling and Analysis 2nd Ed.*. McGraw-Hill 1991.
- [10] <http://mathworld.wolfram.com/MarkovChain.html>.
- [11] Minar, Nelson, R. Burkhart, C. Langton, and M. Askenazi, “The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations,” <http://www.swarm.org/archive/overview.ps> (1996).
- [12] Ross, Sheldon, M, *Simulation 3rd Ed.*. Academic Press, 2002.
- [13] <http://swarm.org>.
- [14] Theodorou, Doros N., “Hierarchical Modelling of Polymers,” Simu Newsletter issue 1, 19–40, March 2000. (Available online at <http://simu.ulb.ac.be/newsletters/newsletter.html>.)

- [15] Wolfram, S., editor *Theory and Application of Cellular Automata*. World Scientific, Singapore, 1986.
- [16] Zeigler, Bernard P., “Hierarchical, modular discrete-event modelling in an object-oriented environment,” *SIMULATION* 49-5, 219-230 (1987).
- [17] Zeigler, Bernard P., “Systems hierarchy as a basis for simulation model description,” *SIMULETTER* Jan, 1984, 15, 1.
- [18] Zeigler, Bernard P., H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems 2nd Ed.*. Academic Press, 2000.