

# AN OBJECT-ORIENTED DESIGN FOR DYNAMIC SIMULATION MODELS

Stephen J. Strommen\*

---

## ABSTRACT

Object-oriented design has evolved as a means for dealing with very complex software systems. This paper outlines some of the fundamental concepts of the object-oriented approach and applies them to the design of a dynamic simulation model of a financial institution. A sample design for a simulation model is presented, including a description of the objects involved and the way they interact.

The paper closes with a discussion of the advantages of object-oriented design, not only in the context of the model presented here but also in the broader context of dynamic modeling in general.

---

## I. INTRODUCTION

Over the last few decades the management of financial institutions has become steadily more complex. The volatility of interest rates has become a significant management issue, and new investment vehicles have been becoming available at an accelerating pace. At the same time, the public has become more educated about the financial options available to them and more ready to exercise them. The proliferation of such options and the use of complex derivative investment vehicles has made it much more difficult to evaluate the financial condition of a financial institution. As a result, actuaries have turned to ever-more-elaborate and complex computer models to help them understand financial risk and manage financial institutions.

At the same time, the software community has seen an increase in the complexity of all types of computer systems. This increase in complexity has paralleled the increase in the power of computer hardware. A number of design approaches have been created to deal with increased complexity of modern software.

Historically speaking, structured design has perhaps been the most influential approach. Under structured design, the task for which a program is designed is divided into subtasks, each of which can be treated as a separate subprogram. Languages such as FORTRAN and COBOL evolved along with structured design.

However, experience has shown that structured design tends to run into difficulty when dealing with very complex systems. Stein (1988) noted that "Structured programming appears to fall apart when applications exceed 100,000 lines or so of code." Modern simulation models with intuitive graphic user interfaces can easily exceed that limit.

Object-oriented design builds upon structured design but introduces several new concepts, including that of an "object," which is different from a subprogram. Under object-oriented design, a computer system is a collection of cooperating objects, each of which is a member of a hierarchy of classes of objects. The concepts of encapsulation, inheritance, and polymorphism are introduced, and they provide very powerful tools for dealing with complexity. More recently developed programming languages such as C++, Object Pascal, and Smalltalk have evolved along with the object-oriented approach.

The aim of this paper is to help actuaries understand the principal concepts of object-oriented design and to show how they can be employed within the context of a simulation model of a financial institution. The hope is that this design approach can help actuaries and software developers deal with the daunting complexity of creating dynamic simulation models of financial institutions.<sup>1</sup>

The remainder of the paper is divided into three main sections. Section II covers the basic concepts of object-oriented design, including encapsulation,

---

\*Stephen J. Strommen, F.S.A., is Associate Director, Financial Planning, Northwestern Mutual Life Insurance, 720 East Wisconsin Ave., Milwaukee, WI 53202.

<sup>1</sup>The design presented in this paper has been implemented and is in use at the author's company for financial modeling work.

inheritance, and polymorphism. Section III, the main body of the paper, introduces a set of objects that can form the core of a dynamic simulation model and explains how they interact. Section IV lists the advantages of the object-oriented approach not only for actuarial models but also for dynamic simulations in general.

## II. CONCEPTS OF OBJECT-ORIENTED DESIGN

### What Is an Object? (Encapsulation)

Most elementary treatments of programming introduce the separate concepts of code and data. Code refers to the computer instructions used to carry out a task, and data is the information used, created, or calculated as part of the task. In a student's earliest classroom exercises, all of the data in a program is accessible by all of the code. Later, if instruction proceeds to structured design, the idea of scope is introduced, which refers to the way some data can be kept within one subprogram and made inaccessible to the rest of a system. However, the separate concepts of code and data remain.

Under object-oriented design, code and data are combined, or encapsulated, into the concept of an *object*. An *object* is a representation of some real-world entity. Every object has a set of *properties*, which are descriptive characteristics of the object. *Properties* are somewhat analogous to data. Every object also has certain actions that it can perform, which are referred to as *methods*. By encapsulating *properties* and *methods* into a single semantic entity, object-oriented design enforces a discipline on scope of access that is missing from basic structured design. Properties (data) cannot be accessed without specifying the object whose property is desired. Methods (code) cannot be executed without specifying which object is to carry out the action.

In the real world, many objects may belong to the same category, or class. A *class* is an abstraction from an individual object and defines what kind of thing the object is. In object-oriented design, the term *class* has the same meaning. The software developer writes code to define a category, or *class*, of objects. In program code, a *class* is defined by its encapsulated set of properties and methods. Every object that is created in a program must be a member of some class. Individual objects of the same class can differ from one another by having different values associated with their properties.

The encapsulation of properties and methods into an object is very powerful because it provides a conceptual "wrapper" around a part of a software system. With such a wrapper in place, the interaction of an object with the rest of the software system can take place only through a well-defined interface. This facilitates the development and use of software components in ways that were never before possible. For example, it is now possible for a user to embed a spreadsheet created using Company A's spreadsheet program into a word processing document created using Company B's word processor. To enable this capability, Company B, which developed the word processor, did not need to know the internal workings of Company A's spreadsheet; Company B only needed to know the interface to the spreadsheet object, that is, its encapsulated set of properties and methods.

To further illustrate encapsulation (and other concepts later on), consider a real-world class of objects: the motor vehicle. While this is not a class that occurs in financial models, it is ideally suited to explaining the concepts of encapsulation, inheritance, and polymorphism, which form the basis of oriented design. To motivate the idea, imagine that you are designing a computer game that simulates and displays the behavior of many motor vehicles on a highway or in a race.

Every motor vehicle must have at least the following properties and methods:

Properties	Methods
Engine size Number of wheels Current speed	Apply accelerator Apply brakes

These properties and methods define the motor vehicle class.<sup>2</sup> There can be several motor vehicle objects in a program. Each one is a member of this class of objects, but has its own copy of the set of properties and methods.<sup>3</sup>

Note that the methods in this example change the state of the object (as represented by its properties). Applying the accelerator increases the current speed, while applying brakes reduces the current speed. This kind of interaction between methods and properties is common.

<sup>2</sup>In a real application the motor vehicle class would have many more properties and methods, but for illustrating concepts we focus on just these few.

<sup>3</sup>When polymorphism does not come into play (as discussed later), most implementations of object-oriented languages keep just one copy of the methods for all objects of the same class. Separate copies of the properties distinguish one object from another.

### Inheritance

The brief set of properties and methods listed above could describe any kind of motor vehicle, from a two-wheeled scooter to an eighteen-wheel trailer truck. Therefore the “motor vehicle” class is an abstract class. A mechanism is needed to add additional properties and methods to an abstract class to define individual real-world objects. In object-oriented design, that mechanism is inheritance.

Inheritance allows one to define a more specific class by inheriting all the properties and methods of an abstract class and then adding some additional properties and methods. Consider two very different kinds of motor vehicles: a convertible sports car and a school bus. Develop a class for each of these two kinds of motor vehicles.

For the convertible sports car class, we inherit the properties and methods of the motor vehicle class and add at least the following property and methods:

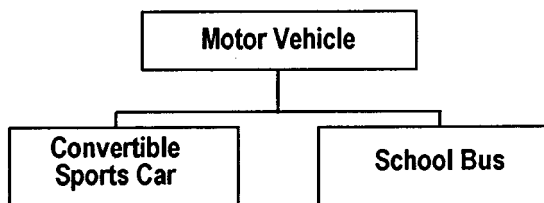
Property	Methods
State of convertible top (up or down)	Put top up Put top down

For the school bus class we inherit the properties and methods of the motor vehicle class and add at least the following property and methods:

Property	Methods
State of flashing warning lights (off or on)	Turn warning lights on Turn warning lights off

We now have three classes, and they form a small hierarchy, as shown in Figure 1. *Motor vehicle* is the abstract class, sometimes called the base class. *Convertible sports car* and *school bus* are sometimes called child classes or derived classes. One can easily envision adding further levels to this hierarchy by defining several different types of school buses, for example.

FIGURE 1  
INHERITANCE



### Polymorphism

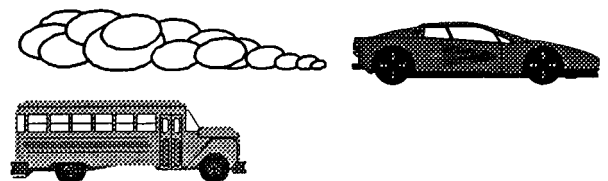
Polymorphism refers to the way two objects of the same abstract class can behave very differently. In our example, a convertible sports car and a school bus each inherited some properties and methods from the motor vehicle class. In particular, they each inherited a method called *apply accelerator*. However, the result of applying the accelerator in a sports car would be very different from doing so in a school bus! The sports car’s current speed property should change more rapidly than that of the school bus.

How can this difference in behavior be achieved, when the sports car and school bus class each inherited the same *apply accelerator* method? It can be achieved by overriding the behavior of the base class in each of the derived classes. In other words, a new version of the *apply accelerator* method is written for the sports car class, and another new version is written for the school bus class. These new methods override or replace the corresponding method of the base class.

To help understand the usefulness of polymorphism, revisit the idea of a game program that displays many motor vehicles on the screen and simulates their behavior. Recall that the class of an object defines the interface between that object and the rest of a software system. The game program must contain a collection of motor vehicle objects, and it deals with each object through the interface to the motor vehicle class. When the program executes the *apply accelerator* method of a motor vehicle object that happens to be a sports car, the *apply accelerator* method of the sports car class is executed, even though the *apply accelerator* method was called through the interface to the motor vehicle class, not the interface to the sports car class. Through this mechanism the motor vehicle class becomes polymorphic in the sense that the same method produces different behavior for different members of the class (see Figure 2).

FIGURE 2  
POLYMORPHISM

Class: Motor vehicle  
Method: Activate accelerator



## Additional Concepts

While encapsulation, inheritance, and polymorphism are the core ideas of object-oriented design, the discussion here has barely scratched the surface of the many methodologies and concepts that fall within the object-oriented paradigm. More complete coverage of the object-oriented approach can be found in Booch (1991) and in Rumbaugh (1991).

### III. AN OBJECT-ORIENTED DESIGN FOR DYNAMIC SIMULATION MODELS

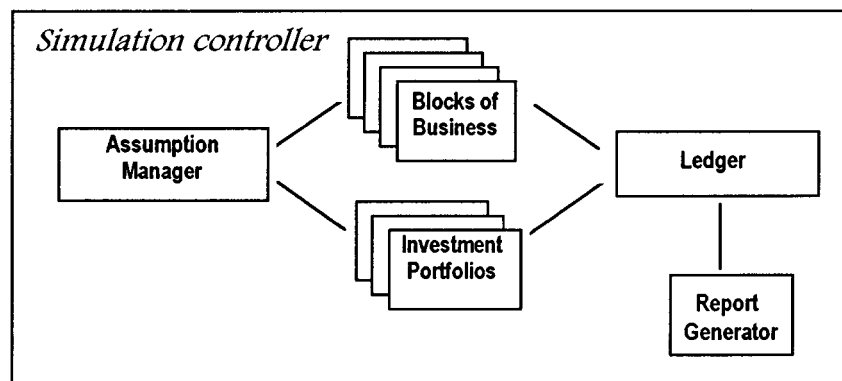
Every dynamic model of a financial institution must contain certain kinds of functionality. By thinking about this at an abstract level, we can begin to identify the principal objects that such a model must contain. Each kind of functionality must correspond to a method of some class of object. Here, then, is a list of the kinds of functionality required in such a model:

- Editing, storing, and retrieving assumptions
- Simulating transactions of insurance policy liabilities (or other financial liabilities)
- Simulating transactions involved in investment management
- Simulating corporate-level items such as taxes, stockholder dividends, and capital transfers
- Recording and storing the results of the simulation
- Generating reports from the stored results.

Each of these areas of functionality is handled by one of the classes shown in Figure 3. That figure provides a top-level view of the structure of the model design that is discussed in more detail below. A few brief comments describing each of the classes provide a perspective on the way in which they relate to one another. Each is discussed in more detail later.

- *Assumption Manager*. The model needs a source of assumptions, and this class provides it. Other objects that carry out simulations will request assumption values from the assumption manager.
- *Blocks of In-Force Business*. A block of business is a set of liabilities that are modeled as an entity. These could be insurance policies in force, checking account balances, or any other kind of financial liability taken on by a financial institution. The block of business class is responsible for carrying out the simulation of those liabilities, including the addition of new sales or additional liabilities to the block. In carrying out the simulation, it will request assumptions from the assumption manager and record results in the ledger.
- *Investment Portfolios*. An investment portfolio is a group of investments that are modeled as entity, with a consistent investment strategy. Financial institutions often have several separately managed investment portfolios, each containing a different mix of investments and having a different investment strategy. In carrying out the simulation, the investment portfolio class will request assumptions from the assumption manager and record results in the ledger.
- *Ledger*. A ledger is very much analogous to the set of accounts actually used in a double-entry accounting system. Simulated transactions are recorded by performing debits and credits to the ledger. However, this ledger also provides a place to record other numerical results such as the face amount of insurance in force or the number of death claims processed. It also maintains snapshots of its state at many points in time, such as calendar year-ends.
- *Report Generator*. The report generator is a utility used to generate reports from the ledger. The report generator provides a means to create virtually any

FIGURE 3  
OBJECTS IN A DYNAMIC SIMULATION MODEL



row-and-column report by specifying the desired row and column headings from information available in the ledger.

- *Simulation Controller*. This is the object that represents the entire financial institution. It contains and manages all the other objects and directs the simulation. It also carries out corporate-level decision-making and simulation of actions such as payment of stockholder dividends.

Keep in mind that object-oriented design is part science and part art. For any reasonably complex software application, there are probably as many different ways to design an object-oriented framework as there are software developers. The design presented here is neither the only possible one nor the only good one. It does, however, provide an implementation approach that makes effective use of the concepts of object-oriented design, including inheritance and polymorphism.

After a brief discussion of how the model simulates and records transactions, each of the classes described above is described individually in more detail.

### How the Simulation Takes Place

To understand the way this model works, it is necessary to understand what it is not as well as what it is. This is not an algebraic model or a trend line model. Models of that sort are typically implemented in spreadsheets. This, rather, is a transaction simulation model. Results are produced by simulating a large number of individual transactions and recording the results in the ledger.

To explain what this means, consider one simple transaction: the payment of a policyowner dividend in a mutual life insurance company. Suppose a dividend of \$100 is paid on a particular policy. Assumptions may indicate that 70% of dividends are applied to the purchase of additional paid-up insurance and the remainder are paid in cash. So \$70 of the dividend is applied to purchase paid-up insurance. Premium rates may indicate that the \$70 purchases \$250 face amount of paid-up insurance for this particular policy.

This transaction would be recorded in two ways, a ledger entry and an update to the state of the in-force block.

Ledger Entry		
Account	Debit	Credit
Dividends paid	100	
Cash		30
Dividends applied to additions (income)		70
In-Force Update		
Add \$250 to the face amount of paid-up additions for this policy.		

In general, transactions result in one or both of the following:

- A set of debits and credits to the ledger
- An update to the state of an in-force block or investment portfolio.

In writing specifications for a model of this type, most of the effort is spent identifying the transactions that will be simulated and specifying the ledger entry and/or data updates used to record each transaction. This paper does not get to that level of detail because it can differ significantly from one implementation to the next. Nevertheless, the framework presented here is designed primarily with the intent of facilitating a transaction simulation methodology.

### The Assumption Manager Class

The assumption manager class provides a means to edit, store, and retrieve assumption values. In financial models most, if not all assumptions are expressed numerically. Typically they fall into two main categories: The first is a single numeric value that optionally changes by calendar date, and the second is a table of values that varies by age and/or duration.

With that in mind, the following set of properties and methods may be all that is needed to provide an interface to the assumption manager class:

- Properties
  - fileName
- Methods
  - value = getValue (assumption name, date)
  - value = getTableValue (table name, age, duration)
  - editAssumptions
  - readFile (file name)
  - writeFile

Note that the first two methods require input arguments (the items described in parentheses) and return some information (the item on the left-hand side of the equal sign). The syntax used above to describe these relationships is used throughout the paper. In addition, names of properties and methods generally consist of two or more words run together (as in getValue). In most programming languages names cannot include blank spaces, so this naming approach is consistent with the implementation of program code.

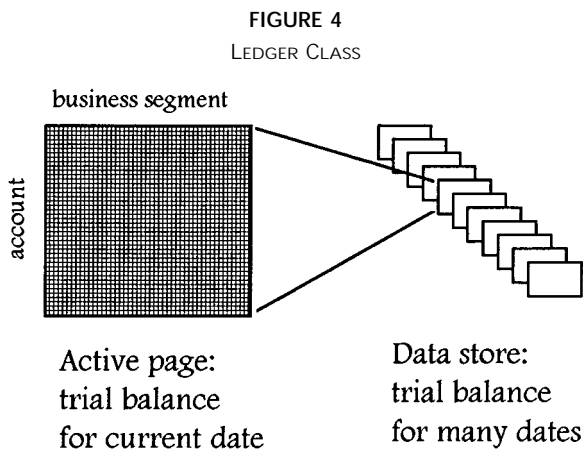
Note that only one file name is used for this class. It is anticipated that all assumptions used in one simulation would be stored in a single file. While this is desirable for purposes of documentation and re-creating results, it may not always be practical. It would be

easy to extend this class to include multiple file names as sources for various kinds of assumptions.

A great deal could be written about ways to implement the functionality implied by this simple set of properties and methods. There are many ways in which the assumptions could be stored, and the user interface for editing assumptions could form the subject of a small book. The intent of this paper is not to delve into these implementation issues. Rather, we stop at this point, because the description of the necessary interface between the assumption manager and the rest of the modeling system is complete.

### The Ledger Class

Figure 4 provides a conceptual view of the *ledger* class. The main body of data managed by the class is a data store containing a set of “pages,” where each page represents a set of account balances set out in rows and columns. Each row corresponds to one account, and each column corresponds to one segment of the company’s business. The set of account balances on each page represents a “snapshot” of the company’s ledger at a single time.



During a simulation, one such page is kept active. Debits and credits are made to this active page throughout the simulation. Copies of the active page are made and placed in the data store when the simulation reaches the end of a time period, such as the end of a month or a year. In this way, a ledger object always contains not only current ledger balances but also copies of past ledger balances at periodic calendar intervals.

Here is the list of properties and methods needed to provide an interface to the ledger class:

- Properties
  - fileName
  - lists of accounts, business segments, and dates
  - currentDate
- Methods
  - debit (account name, business segment, amount)
  - credit (account name, business segment, amount)
  - value = getBalance (account name, business segment)
  - close
  - store
  - retrieve (date)
  - readFile (fileName)
  - writeFile

Several of these properties and methods require special comment.

The *fileName* property indicates the name of the file where the data store exists and/or is written at the end of a model simulation. This file must contain not only the account balances but also the lists of accounts, business segments, and dates that describe the contents of the data store.

The lists of accounts and business segments can be more than simple lists. They can (should) contain what might be called “virtual” members, which are formulas defined by using the “real” members of the list as variables. For example, a virtual account called “net operating gain” could be defined as a formula based on all income and disbursement accounts. A virtual business segment called “total company” could be defined as the sum of all individual business segments. The use of such virtual members in these lists makes it possible for a ledger object to return a value for total company net operating gain with a single call to the *getBalance* method.

The *debit* method simply adds an amount to the appropriate slot in the active page. The *credit* method subtracts an amount from the indicated slot on the active page. Credit balances are stored as negative numbers and debit balances are stored as positive numbers. This is a common convention in accounting systems.

The *close* method sets all income and disbursement accounts in the active page to zero. This is commonly done at the end of each fiscal year. Implementation of this method requires each account to be tagged as to its account type, so that income and disbursement accounts can be identified separately from other accounts.

The *store* method is used to copy the active ledger page to the data store. When this is done, the *currentDate* property determines which time slot in the data store is used.

The *retrieve* method is the reverse of the *store* method. It copies a ledger page from the data store to the active page. The date provided as an input argument determines which page is retrieved and is then used to set the *currentDate* property.

Note that the ledger in a model like this can store numerical results that would not normally find their way into a real ledger system. Data such as the face amount of insurance in force or the number of policies issued can be stored in the ledger. Accounts that store such data should be tagged as of a special type so that they can be excluded from any test to determine whether the ledger balances (that is, the debits equal the credits).

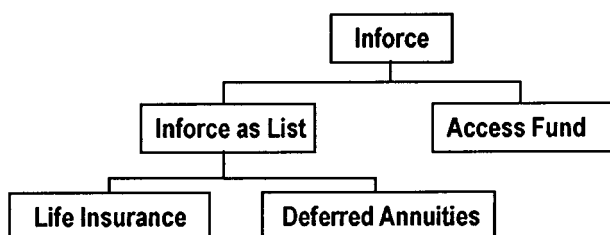
Note that the interface to the ledger class described here is independent of the way the data store is managed. It could be kept in a file, with only the active page in core memory, or it could be kept entirely in memory. It could be represented using a database structure, or it could be represented as a simple three-dimensional array. In fact, a ledger class using one data store implementation could be substituted for a ledger class using a different data store implementation without affecting the rest of the model's operation in any way. This kind of independence between the interface and the implementation of a class is an important advantage of object-oriented design.

### The In-Force Block Class

The *in-force block* class represents a block of business in force, such as life insurance policies, annuities, or (in a bank) savings deposits. An *in-force block* can be thought of as a line of business, except that it embodies only the liability portion of a line of business.

The *in-force block* class is an abstract class and forms the base class of a hierarchy of classes. Both inheritance and polymorphism come into play in building such a hierarchy. Figure 5 shows an example hierarchy that could be built.

FIGURE 5  
IN-FORCE BLOCK CLASS



In Figure 5, two classes are derived directly from the abstract class. One represents an *access fund*, which is a block of money market accounts with checking privileges. A life insurance company may maintain such a block of business as a service for beneficiaries of death claims, who need a place to put their funds where they are safely invested and readily accessible while family and financial arrangements are made. Such business can be modeled very simply by accumulating the aggregate of all account balances at a money market interest rate and adjusting the aggregate balance for transfers in and out of the fund.

The second class derived directly from the abstract class is labeled *in-force as list*. In this class, the business in force is represented as a list of individual policies or groups of policies. This is another abstract base class, because the code needed to loop through a list of policies is implemented, but the processing of each policy in the list is not.

The last two classes in Figure 5 are *life insurance* and *deferred annuities*. These classes inherit the functionality of the *in-force as list* class and add the capability of processing the individual policies.

With this hierarchy in mind, the following is a list of properties and methods that are needed in the top level abstract class:

- Properties
  - associated assumption manager
  - associated ledger
  - currentDate
- Methods
  - cashflow = processMonth
  - performValuation
  - value = getTotalAmount (type of total)

In order for an *in-force block* class to carry out its processing, it needs to know where to obtain assumptions and where to store results. Therefore it must have properties that provide that information. In addition, since an object of the *in-force block* class represents the state of some business on a particular date, that date must be a property of the class.

The *processMonth* method carries out the simulation and recording of all transactions for one month. This results in ledger entries and changes to the state of the object itself. This is a key point where polymorphism comes into play. When a simulation model calls the *processMonth* method of an object descended from *in-force block*, the processing that takes place varies depending on the class of the object, that is, the kind of business that is being modeled. If several different objects descended from *in-force block* are in the model, each will be processed differently, as appropriate.

The *processMonth* method returns a value equal to the net cash flow generated by the processing for the month. This comes into play later when the connection between assets and liabilities is discussed.

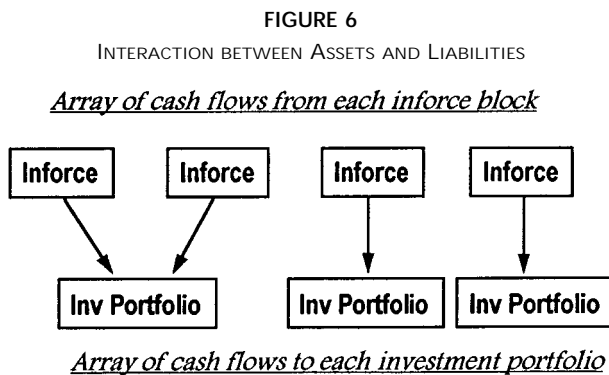
The *performValuation* method is needed to provide a way to request the calculation and recording of period-end balance sheet items like reserves for insurance business. The *performValuation* function should also record income or disbursement accruals that are based on the valuation, such as the increase in reserves.

The *getTotalAmount* method provides a means of querying the object about its total size. Size can be measured in many ways, such as number of insurance policies or amount of insurance in force. Therefore *getTotalAmount* requires an input argument that specifies which measure is desired, and it returns a value that is the total of the requested measure.

Interaction between Assets and Liabilities

As noted above, the *processMonth* method of an *in-force block* returns a value that is the net cash flow generated by all the liability transactions. A simulation model of a multiline financial institution can contain several *in-force blocks* and several investment portfolios, but not necessarily an equal number of each. A means is needed to keep track of the amount of cash flow coming from each *in-force block* and the amount that should be allocated to each investment portfolio.

Figure 6 illustrates the situation. Each *in-force block* produces cash flow, and each investment portfolio receives cash flow. In Figure 6, two of the *in-force blocks* share the same investment portfolio, while each of the other *in-force blocks* is associated with its own separate investment portfolio.



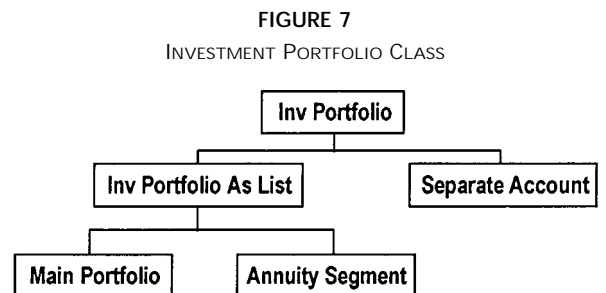
To keep track of the cash flows generated by each *in-force block* and the cash flows going into each investment portfolio, two arrays need to be maintained by the model. In one array, each element is the cash flow generated by an *in-force block*. In the other array each element is the cash flow to be allocated to an investment portfolio. The method of allocation between these two arrays will be specific to each model implementation, but the need for the separate arrays of cash flows is always present.

As will be seen later, the cash-flow arrays will be properties of the simulation controller class, and the allocation of cash flows will be a method of that class.

The Investment Portfolio Class

The *investment portfolio* class represents a group of investments that are managed as an entity with one investment strategy. Except for that, this class is very much analogous to the *in-force block* class. Like *in-force block*, this is an abstract class that forms the base of a hierarchy of classes. Inheritance and polymorphism again come into play in building the hierarchy.

Figure 7 shows an example hierarchy that could be built. Two classes descend directly from the base class. One is a “separate account,” which in the U.S. is often a portfolio of equity investments held at market value. In a very simple model a separate account could be modeled as simply an aggregate market value that changes over time based on an assumed total return and assumed cash flows in and out.



The second class derived directly from the abstract class is labeled *portfolio as list*. In this class, the investment portfolio is represented as a list of individual investments. This is another abstract base class, because the code needed to loop through a list of investments is implemented, but the processing of each investment in the list is not.

The last two classes in Figure 7 are *general account* and *annuity segment*. These classes inherit the functionality of the *portfolio as list* class and add the capability of processing the individual investments and simulating a specific investment strategy.

The following are the properties and methods needed in the abstract *investment portfolio* class:

- Properties
  - associated assumption manager
  - associated ledger
  - currentDate
  - cashBalance
- Methods
  - processMonth (cash flow)
  - performValuation
  - addToCashBalance (amount)

As noted before, these properties and methods are mostly analogous to those in the *in-force block* class. However, there are important differences.

First, every investment portfolio must maintain a cash balance, so the cash balance must be a property of the class. Also, a method is needed to move cash in and out of the portfolio. Here it is called *addToCashBalance* and takes an argument that is the amount to add (or subtract if it is negative).

Another important difference from *in-force block* is that the *processMonth* method takes an input argument that is the amount of cash to add to the portfolio and does not return a value.

The *performValuation* method is needed to calculate and record the market value of investments that may be valued at amortized cost or on other bases in the ledger. In sophisticated models, the *performValuation* method could also calculate and record descriptive statistics such as the duration and convexity of a fixed-income portfolio.

### The Simulation Controller Class

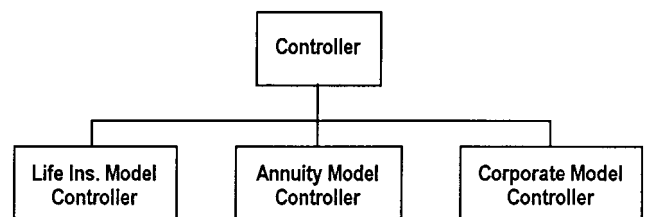
The simulation controller class can be thought of as the class that represents the entire company. The simulation controller contains all the objects discussed so far, including the in-force blocks and investment portfolios, the assumption manager and ledger. The purpose of the simulation controller is to oversee the entire simulation process and take care of corporate-level items such as capital transfers and tax payments. To carry out this purpose, the class needs to provide only one method in its interface, the method that runs the entire simulation.

Depending on the purpose of the model, the simulation controller may be very simple or very complex.

Therefore, it is an abstract class that forms the base of a hierarchy of classes. Inheritance and polymorphism again come into play in building the hierarchy.

Figure 8 shows a sample hierarchy that could be built. In this case the hierarchy is very flat, with only one level of inheritance. The simulation controller appropriate for most models, whether for one line of business or for a multiline company, can be derived directly from the abstract base class. However, the base class can provide the system interface.

FIGURE 8  
SIMULATION CONTROLLER CLASS



The following are the properties and methods of the *simulation controller* class:

- Properties
  - an assumption manager
  - a list of Inforce Block objects
  - a list of Investment Portfolio objects
  - a ledger object
  - cash-flow allocation arrays
  - currentDate
  - run-time parameters
- Methods
  - runCalculations
  - other methods called by runCalculations

The first four properties are objects that have been previously discussed. Note, however, that we have a *list* of in-force blocks and a *list* of investment portfolios. The use of these lists gives great flexibility to this framework, enabling it to handle anything from a single insurance policy to a full multiline company.

The cash-flow allocation arrays were discussed earlier in the paper. During the simulation for each time period, the elements of one array are filled with the cash flow coming in from each in-force block, and the elements of the other array are filled with the cash flow allocated for investment in each investment portfolio. A method called *allocateCashFlow*, which is called from *runCalculations*, is responsible for the allocation of cash flows between the two arrays.

Since the simulation controller provides a representation of the company at a given time, the current date is needed as a property.

When a simulation is to be run, the user must specify parameters such as the starting date, length of time to simulate, and the names of files in which various kinds of input data are stored. This kind of information can be collected into a set of run-time parameters and must be available to the simulation controller. It can be useful to design a separate class to represent a collection of run-time parameters, but that is not discussed further here.

The *runCalculations* method forms the core of the simulation controller class. The *runCalculations* method calls several other more specific methods in order to carry out its work. Each such method is discussed below, grouped under the three main phases of the processing: initialization, simulation, and saving results.

#### Initialization

- *SetupObjects* (\*) creates the assumption manager, in-force blocks, and so on in the memory of the computer. It also initializes these objects using information obtained from the files named in the run-time parameters. A special version of this method is needed for each model implementation.
- *PerformValuation* initializes the balance sheet by calling the *performValuation* method of each in-force block and each investment portfolio. Since those objects are polymorphic, the appropriate kinds of valuation calculations are done for each.
- *CustomSetup* (\*) allows for any special corporate-level initialization that may need to be done. A special version of this method is needed for each model implementation.

#### Simulation (a Monthly Loop through the Following Methods)

- *ProcessInforces* calls the *processMonth* method of each in-force block and fills one of the cash flow arrays with the values returned from *processMonth*. Since the in-force blocks are polymorphic, the appropriate processing is carried out for each.
- *AllocateCashFlow* (\*) allocates the amounts from the cash flow array set up in the previous step to the cash flow array that represents the amount to be added to each investment portfolio. A special version of this method is needed for each model implementation.

- *ProcessInvestments* calls the *processMonth* method of each investment portfolio. Since the investment portfolios are polymorphic, the appropriate processing is carried out for each.
- *PerformValuation* calls the *performValuation* method of each in-force block and each investment portfolio. Polymorphism again ensures that the appropriate valuation calculations are done in each area.
- *PeriodEndProcessing* (\*) performs any corporate-level processing that may need to be done at the end of each period, such as payment of dividends, calculation of tax liabilities, or allocation of investment income. A special version of this method is needed for each model implementation.

#### Saving Results

- *SaveResults* calls the *writeFile* method of the ledger to save the results of the simulation. It also takes care of clearing the computer's memory of the objects used in carrying out the simulation that are no longer needed.

An asterisk (\*) was used in the list of methods above to indicate those methods for which a special version must be written for each model implementation. For each such implementation, a new class would need to be created by inheriting everything in the *Simulation Controller* class and then overriding the starred (\*) methods as needed.

#### The Report Generator Class

The report generator class provides a parameterized way to specify and generate reports containing data from the ledger. It is a utility class and has no role in actually carrying out the simulation. Therefore the detail of its properties and methods is not discussed here. The important idea is the conceptual approach to parameterizing the definition of a report.

The data stored in a ledger object can be thought of as a cube. The data has three dimensions: date, account, and business segment. Given a value for each of those dimensions, an account balance can be retrieved from a ledger.

A report in row-and-column format can be thought of as a two-dimensional slice through the three-dimensional cube. A single value is chosen along one dimension, and a slice through the cube is made orthogonally at that value. The slice represents a two-dimensional matrix of values. Along one of these dimensions lay the choices for the row headings of a

report; along the other lay the choices for column headings; and the values in the matrix provide the data for the report.

Given this procedure, a great variety of ad hoc reports can be specified in parameterized form. Consider the following:

- *A statement of operating income for one period, broken down by business segment.* To parameterize this report, specify one date, a list of accounts (for the line items of the report), and a list of all the business segments. Also, specify that the first list is for row headings and the second list is for column headings.
- *A statement of total company operating income for each projected period.* To parameterize this report, specify the total company as the business segment, and provide a list of accounts and a list of all the dates. Also, specify that the first list is for row headings and the second list is for column headings.
- *A breakdown of premium from new sales by date and business segment.* To parameterize this report, specify the account used for new premium, a list of business segments, and a list of all the dates. Also, specify that the first list is for row headings and the second list is for column headings.

Note that this simple description leaves out some important points. Consider a statement of operating income. The rows in such a report are not all simple account balances from a ledger, one after the other. Some rows are totals, such as total premiums or total expenses. Also, some of the rows are blank or underlined.

The capability to include totals such as total premiums or total expenses makes use of the “virtual account” capability of the ledger, which was discussed earlier. Such totals can be defined as virtual accounts. That way the report generator does not need to “know” anything about how to add up the totals; the ledger simply provides them. Virtual items can be included in the lists of accounts and/or business segments used to define a report.

The capability to include blank lines and underlined items can be implemented by allowing lists to include blank items and by tagging each element of a list with data regarding its appearance (underlined, boldface, and so on).

With these capabilities in place, it is possible to provide a user with the means to easily generate a wide variety of ad hoc reports. In addition, pre-defined reports can be stored in parameterized form and produced from any previously stored ledger on request.

#### IV. ADVANTAGES OF THE OBJECT-ORIENTED APPROACH

The object-oriented approach was introduced in Section I principally as a means to help deal with the complexity of modern simulation models. However, there are many other advantages that accrue to the combination of the object-oriented paradigm and the specific design presented here. These include:

- *Clear implementation of cash-flow modeling.* This design clearly implements the cash-flow interaction between assets and liabilities. By processing both assets and liabilities for one time period before proceeding to the next time period, this design provides an ideal platform for reflecting the dynamic behaviors that result from volatile asset and liability cash flows.
- *Ability to write specifications at an abstract level.* Most actuaries are not specialists in software design and implementation. Actuaries and systems developers can use the design at the level of abstraction presented here as a common frame of reference. Because most of the processing involves the simulation of transactions, actuaries which develop specifications can focus on specifying what transactions take place and what ledger entries and data updates are needed to record each transaction.
- *Implementation in small, independent pieces.* Once the interface to a class is defined, the implementation can be carried out without the need for a global understanding of the entire system. This allows the software developer to focus clearly on the task at hand and simplifies that task greatly. This is one of the primary ways in which object-oriented design helps deal with the complexity of modern simulation models.
- *Re-use of utility classes.* Many of the classes discussed here are common to many simulation models. These include the assumption manager, the ledger, the report generator, and even some investment portfolios. The ability to re-use these classes greatly speeds up the process of developing a model for a new product, line of business, or type of financial institution.
- *Ease of adding additional decision criteria.* Additional corporate decision-making criteria and management responses can easily be added to a model in this design using inheritance. The developer can inherit all the functionality of the existing simulation controller and then add whatever else is desired.

Flexibility is another key attribute of this approach to an object-oriented design. Flexibility is provided in all the following ways:

- *Applicable to any financial institution.* While the discussion here has been mostly in the context of a life insurance company, the liabilities modeled could include anything from automobile insurance policies to bank savings and checking accounts. A dynamic model of nearly any financial institution can be implemented by using the same general framework.
- *Scalability.* The design presented here can scale all the way from a single insurance policy to a multiline financial institution. This provides flexibility for handling a wide variety of modeling needs.
- *Stochastic capability.* Nothing in this design prevents a simulation from being carried out in stochastic fashion. Stochastic processing could be used with respect to either assets or liabilities or both. Although the design as presented so far provides an engine for running a single scenario, it would be a simple task to call the engine multiple times to produce multiple stochastically generated sets of results.

In summary, object-oriented design is appropriate for dynamic simulation models of all kinds. An analysis of the objects involved in a process and the ways in which they interact is the first step in the development of any dynamic model. Because actuaries are commonly called upon to evaluate and manage situations that involve dynamic behavior, an understanding

of object-oriented analysis can be helpful in many kinds of actuarial work.

Research and development of the object-oriented paradigm is ongoing. There is an established literature, as reflected in the extensive bibliography provided in Booch (1991). There are also ongoing efforts to develop and standardize a graphic notation for representing the relationships between classes in an object-oriented design. Current efforts towards such a "Uniform Modeling Language" are documented on the internet at <http://www.rational.com/uml/index.html>. Actuaries involved with modeling work will find it beneficial to obtain an understanding of the work taking place in this field.

#### REFERENCES

- BOOCH, G. 1991. *Object Oriented Design with Applications*. Redwood City, Calif.: The Benjamin/Cummings Publishing Company.
- RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.
- STEIN, J. 1988. "Object-Oriented Programming and Database Design," *Dr. Dobb's Journal of Software Tools for the Professional Programmer*, No. 137 (March): 18.

*Discussions on this paper will be accepted until January 1, 1998. The author reserves the right to reply to any discussion. See the Table of Contents page for detailed instructions on the preparation of discussions.*