

*On Hierarchy of Actuarial Objects:
Data Processing from the
Actuarial Point of View*

Aleksey S. Popelyukhin, Ph.D.

On Hierarchy of Actuarial Objects: Data Processing from the Actuarial Point of View

Aleksey S. Popelyukhin, Ph.D.

Introduction

Like all professionals in the information era, actuaries need computers to automate non-creative activities and to relieve them from the burden of repetitive actions.

Actuaries need a system which shields them from the complexities of computer architecture and provides an abstraction and generalization exactly at the level of the common denominator of all actuarial functions.

From the actuarial point of view, an ideal data processing solution is a (a) transparent to users (b) highly efficient (c) storage/retrieval system for (d) structured actuarial data (objects) with (e) an extremely flexible (f) computationally complete (g) open (h) calculation engine. In short, a system which speaks actuarial language and makes it very easy to express actuarial algorithms and very hard to make mistakes. The paradigm where goals of abstraction, flexibility, simplicity and reliability can easily be achieved is the Object-Oriented (OO) model.

In order to "teach" an object-oriented data processing system to "speak actuarese" actuaries need to structure and categorize their data as well as formalize their algorithms. A well-defined hierarchy of actuarial objects creates an environment for the effortless expression of actuarial business rules and algorithms.

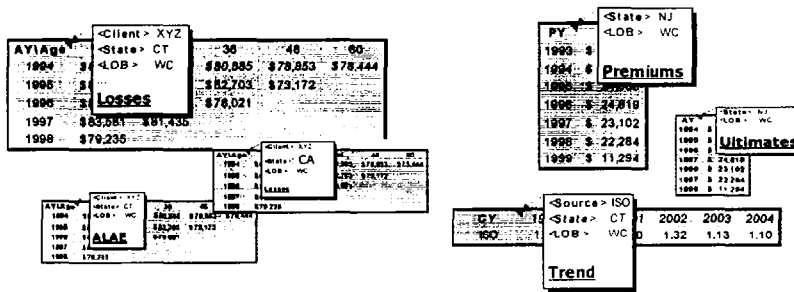


Figure 1

To perform their professional duties, actuaries operate with chunks of structured data, each chunk with its own set of properties (see Figure 1.) Some properties (line of business, location) help to distinguish one chunk of data from another, while other properties (loss vs. ALAE, dollars vs. counts) describe "actuarial nature" of the data and help determine which actuarial operation is appropriate to perform on them. It is intuitively clear that different kinds of properties differ in

their origin and their effects on actuarial calculations. It is also immediately apparent that proper use of these properties in the actuarial data processing system may significantly increase the system's effectiveness and significantly reduce mismatch between data chunks and the algorithms applied to them. Let us formalize these findings and make evident that the distinction between different kinds of properties lies as deep and is as fundamental as the difference between *object* categorization and *class* hierarchy in an object-oriented model. Let us also demonstrate how this knowledge can be communicated to OO system designers and used to build effective and reliable actuarial data processing solutions.

Object Orientation*

Object Orientation is a preferable paradigm for

- *real world modeling*
- *creation of reusable, extendable and maintainable software components*
- *construction of reliable and consistent applications*

The OO paradigm facilitates communication between the user/actuary and the system designer. For example, compare the same calculation expressed in spreadsheet syntax and in OO fashion.

<code>= (sum (C35:C39) -max (C35:C39) -min (C35:C39)) /3</code>	<i>(Spreadsheet)</i>
<code>AgeToAgeFactors.Average (Type:=ExclHiLo, LastDiagonals:=5)</code>	<i>(OO)</i>

Figure 2

The former expression does not communicate to the user the purpose of calculation, and is prone to errors. Nor is it the best possible algorithm: indeed, it requires 3 passes through the array C35:C39 (for *sum*, *max* and *min*) instead of single pass. On the other hand, as latter expression demonstrates, OO approach creates an intuitive environment for the user (when he needs an average, he just requests so) and leaves the freedom of implementation to the system designer. When an algorithm gets updated due to improvements or error corrections, user's code remains intact contributing to consistent and self-documenting actuarial application.

In a properly designed OO application, the only way to manipulate the data encapsulated within the object is by calling methods of (sending messages to) that object. Not only does such an approach protect data, maintaining the whole data structure in-sync, it also contributes to

- *Reusability: all the code needed to manipulate the data is contained in a detachable module.*
- *Maintainability and extensibility: all fixes and improvements can be made in a very localized place.*

* An excellent introduction to OO concepts and methods can be found in [1]. Martin [2] in a highly conceptual fashion discusses the theoretical foundation of OO technology, while [3] - [5] fully cover subject of OO databases.

- *Usability: hiding implementation of the methods and complexity of the data structure, the OO design provides means for proper and effective use of objects.*

The central notion in the object-oriented model is (surprise!) an object^{*} – an entity, which contains both structured data elements (properties/attributes) and code (methods/operations) - for manipulations with the data. A set of objects with the same structure and behavior is declared and implemented through classes. Class contains both the description of the data structure and implementation of the methods. Thus, class is implementation of the object, while object is an instance of the class.

In order to model complexity of real life objects and variety of their relationships, OO approach relies on

- *encapsulation (data hiding and abstraction).*
- *inheritance (likeness) and*
- *polymorphism (overloading)*

Encapsulation is a mechanism of binding data and operations on that data into single entity. One cannot access encapsulated data directly – all the manipulations on the data are done exclusively through operations associated with the data. Encapsulation, as a way to hide (and, thus, protect) data and privatize (and, thus, abstract) implementation of the object's behavior, shields the user from the object's internal complexity and allows operations with objects as whole entities rather than fractional structures.

Inheritance is a mechanism that facilitates the reuse of the program code from class to its ascendants (subclasses). Through this "class-subclass" relationship, inheritance naturally imposes a hierarchical structure on the collection of the classes. Inheritance, as a way to model "is like" relationship between objects, provides users with the ability to express structure and behavior of complex objects through the simpler ones and, on the other hand, reuse the code and derive new objects from the existing ones.

Polymorphism is a mechanism for declaring multiple operations with same name applicable to arguments of different types. Polymorphism models our real life ability to notice similarities between actions on different types of objects and our desire to use the same verb to name these actions. Polymorphism, as a way to apply the same operation to different classes of objects, contributes heavily to the generalization of algorithms and, thus, helps to avoid unnecessary repetition and duplication of errors.

Example 1. For illustration, let's consider an actuarial triangle as an object. A triangle is the most intriguing actuarial object and the quality of its implementation may greatly affect the effectiveness of the whole actuarial system. Let's start with the storage structure. While it is most intuitive to store elements of the triangle as cells of the encompassing two-dimensional matrix, it may be not the best approach: first of all, almost half of the storage space would be wasted on empty cells and, secondly, not

^{*} Authors of different books on OO subjects define major OO terms somewhat differently. "Object technology has its own vocabulary, which is *large* and *complex*. In its present state, it is unfortunately also *inconsistent*" (see [6]). For precision we cite exact definitions from [6] in the Glossary section.

every computer language and development environment supports variable size (dynamic) 2-D arrays. The most economical way to store a triangle would be Cantor-inspired enumeration of its elements into one-dimensional array (see Figure 3.) :

*element (i, j) of the triangle maps into element $k = (i + j - 2)(i + j - 1)/2 + i$ of the 1-D array**

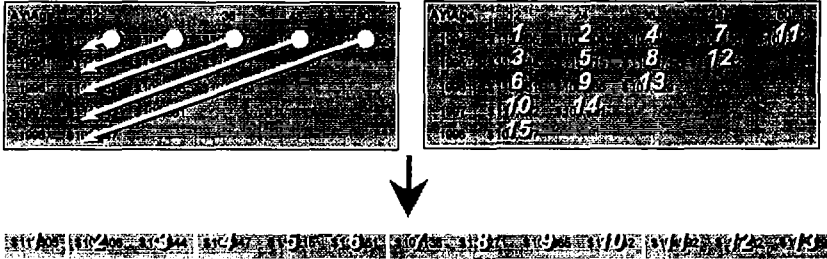


Figure 3

Not only does such a technique yield the most space-conscious arrangement of triangle's elements, it also provides an opportunity to place a whole triangle as a single record in the database, and it makes appending (and extraction) of the last diagonal as trivial as adding (reading) several consecutive elements at the end of the array.

Thanks to encapsulation, as long as in response for the message "RetrieveTriangle" our object will return a familiarly looking half-empty matrix, user won't notice that elements of the triangle are stored in somewhat unusual way. Thanks to inheritance, we may derive different classes of triangles (like those with missing first diagonals, or those with only integer elements for representing "counts") without rewriting mapping formulas. And thanks to polymorphism, we may need to implement some basic manipulations on the triangles (like addition or trending) only once despite the existence of several different classes of triangles.

The natural desire to store objects in some organized fashion triggered the development of OO Databases. OO Databases introduced such fundamental notions as Persistence and Identity.

Persistence refers to availability of the objects across executions. Unlike temporary variables in the computer memory, persistent objects do not disappear when the program stops – they are stored for the future access.

Identity is a mechanism for distinguishing objects and a guarantee for their uniqueness. To insure uniqueness OO databases rely on the object identifiers (OID.) – values, which are unique, permanent and indifferent to the properties of the object. A good example of OID is a Social Security Number: it is unique, permanent and indifferent to the owner – one cannot describe a person looking just at SSN. In real life, however, we do not use an OID for identifying an object,

* For "non-isosceles" triangles $k = \text{ceil}((i * \text{slope} + j - \text{slope} - 1)(i * \text{slope} + j - 1) / (2 * \text{slope}))$, where *slope* is the ratio of interval between rows over interval between columns and $\text{ceil}(a)$ is a minimal integer not smaller than *a*.

rather we use a list of properties to describe the object we want – e.g., name, age and address which are properties of the “person” object and the most used identifiers, but not a person’s OID.

Object Categories

No system can be called object-oriented unless it supports data encapsulation, inheritance and polymorphism. OO databases add requirements for object persistency and identity. Inheritance and polymorphism call for *class* (“internal”) hierarchy, while the identity required by an OO database calls for *object* (“external”) hierarchy!

Every insurance/re-insurance company has amassed a set of actuarial data arrays (triangles, rows, columns, diagonals, etc.) and preferred actuarial analysis techniques. Availability of established sets of actuarial categories and algorithms both simplifies and complicates OO Analysis and OO Design procedures for the OO actuarial data processing system. Simplification comes from the fact that most of the existing categories can probably be reused in the OO hierarchy and many of the algorithms can probably be wrapped into OO functional classes. Complications arise when OO Design requirements demand new categories to be introduced (or existing ones to be reshuffled) and algorithms to be adapted for the newly established object classification.

Every time an actuary attempts to tell apart different data arrays he has to introduce a (or use an existing) category with members describing these data arrays properties. Any distinction, which contributes to the criteria of identity (i.e., every property, which helps to distinguish one data array from others), generates new category or new member of an existing category. Category/member structure applied to the universe of all data arrays is called classification.

It is crucial to realize that an existing data array can be considered as the data portion of an actuarial object, and that an object also may store (among other things) information about what member of which category this object is. In essence, one can think of an actuarial object as a matrix with genealogy, or even simpler, “a triangle, who knows who he is” (see Figure 4.)

AYAge		36	48	60
1994	\$	107,847	\$ 115,288	\$ 124,592
...				
1995	\$ Shape -->	Triangle	110,271	\$ 112,562
1996	\$ Amount -->	Losses	104,029	
1997	\$ Cumulative -	True		
1998	\$	105,847		

Figure 4

Not all categories were created equal. While some categories reflect an “actuarial nature” of the object, others are used just to distinguish similar objects of the same “nature”.

There are 4 major kinds of categories:

1. Those which define an object's place in a class hierarchy (class attributes)
2. Those which define an object's state
3. Those which serve identification purposes (dimensions^{*})
4. Those used for grouping within dimension (generations)

A good example of the 1st kind of category would be "Shape." Indeed, members of this category belong to different classes, possibly inherited one from another: a Triangle (a member of this category) is a Matrix (another member) with half of the cells being empty and some additional specific functionality discussed below, a Diagonal (one more member) is a Triangle with even more empty cells and some more specific functionality, etc... A category "AccumulationType" would perfectly illustrate the 2nd kind of categories: members of this category (Cumulative, Incremental) define an object's state. "Line of Business" and "Location" are primary examples of the 3rd kind of categories, while "Groups of LOB's" and "Regions" with members like "All Liability", "All Property", "NorthEast" and "South West" perfectly represent the 4th kind of categories.

Regions		LOB		
	States	AL	GL	WC
		UW Yrs		
NorthEast	CT	1998	☐	☐
		1998	☐	☐
	NJ	1997	☐	☐
		1998	☐	☐
	South	KS	1998	☐
1997			☐	☐
TX	1997	☐	☐	☐
	1998	☐	☐	☐

Figure 5

Categories of the first two kinds affect the way calculations are performed on the object's data, and thus affect object behavior; they reflect the inner actuarial "nature" of the object and in that sense they belong to the "internal" hierarchy. The remaining categories are imposed by the database requirement, which calls for every object to be uniquely identified; they describe an

^{*} See Figure 5. For precise definitions of *dimensions*, *generations* and *members of dimension* see Glossary.

object and affect the structure of the external (relative to the object) entity - an OO database - and in that sense they belong to "external" hierarchy.

To summarize, classification of the *objects* (within a class) serves two main purposes: identification and selection in OO database, while *generations* provide convenient means for grouping. This is significantly different from the purposes of the *class* hierarchy, which defines inheritance and affects behavior of the objects.

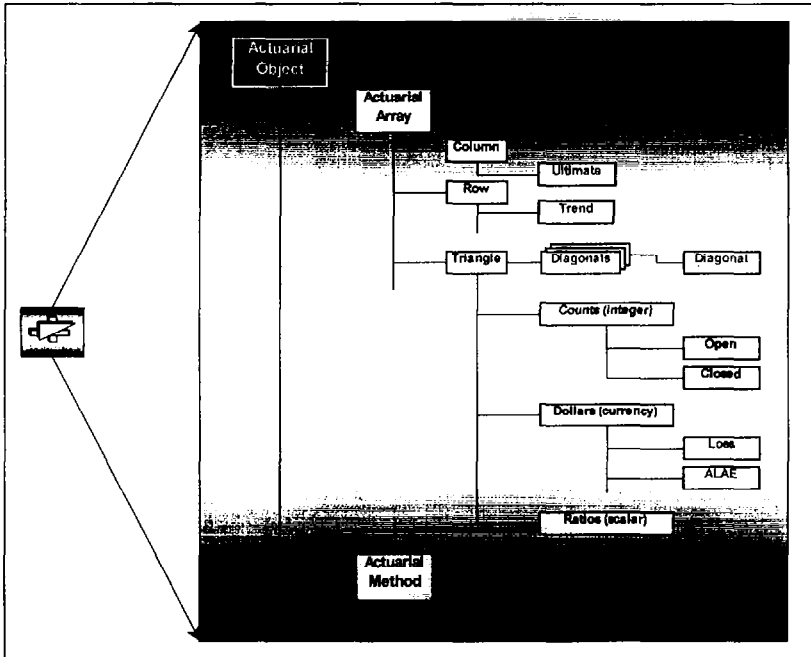


Figure 6*

The internal hierarchy includes categories which affect and are affected by the algorithms. The external hierarchy is the set of all objects factorized by internal hierarchy. Factorization is similar to packing items into the bags: each bag may contain several items, possibly, with their own classification, but factorization helps to classify bags themselves, ignoring what's inside (see Figures 5-6).

To build an OO data processing system, actuaries, during the OO Analysis stage of development, have to clearly define and segregate all 4 kinds of categories. It is important to realize that a

* There exist many different notations for expressing relationships between classes: Booch, Rumbaugh, OMT and, most notably, Universal Modeling Language UML (see [7], [8]). But for Figure 6 we used none of them, because Figs. 5-6 illustrate the notion of factorization rather than a particular OO design.

decision to place a category into an internal or external category will deeply affect the architecture and functionality of the resulting OO system. There is no single recipe for all companies: the same category could be internal in one company, external in another one and not exist at all in the third one. What is true for every company, however, is the fact that the classification can not be designed separately from the algorithms collection!

1st rule of thumb. To determine which hierarchy (internal or external) an actuarial category belongs to, one should take into account the following considerations:

- (a) *whether or not different members of this category need different algorithms to process them ("Counts" and "Dollars" as members of the "Amounts" category usually need different algorithms, while "NY," "NJ" and "CT" as members of the "Location" category are usually treated the same way),*
- (b) *whether or not different members of this category affect the way algorithms are applied (the "Cumulative" and "Incremental" members of the "AccumulationType" category require somewhat different calculations),*
- (c) *whether or not members of the category are used to define groups for possible aggregation into subtotals (the "NorthEast region" and "SouthWest region" members of "Regions" category can be defined through the groups of members from another category "Locations" and they do not serve identification purposes directly).*

Categories for external hierarchy should be defined in such a way, that two main activities – selection and aggregation (grouping) – be optimized. This approach may help to eliminate unnecessary levels in the hierarchy. If there is no intent to summarize amounts (data or results) across the members of a particular category, it may be blended with other categories, thus simplifying hierarchy. For example, the categories "Line of business" and "Sub-line" can be combined for something like {"Fire", "WC Med", "WC Ind", "GL BI", "GL PD"}. However, if category members simplify the selection process, then a category should be created. For example, category "DAC" with members {"Direct", "Assumed", "Ceded"} may significantly simplify selection of objects for "Gross vs. Net" actuarial analysis.

Another consideration for determining categories serving as dimensions in an external hierarchy is density. A multi-dimensional array is dense (as opposed to sparse) if a relatively high percentage of the possible combinations of its dimension members contain data values. Some categories may be combined in order to avoid impossible combinations of its members. For example, if only few lines of business have tail coverage, it make perfect sense to combine the "Line of Business" category with the "Tail Indicator" category (unless, of course, there are special algorithms for processing lines with tail coverages: in that case the "Tail Indicator" category belongs to internal hierarchy).

The analogy with currently available actuarial systems lies in the fact that sets of existing spreadsheets (different for different data types) are roughly equivalent to the categories of the 1st kind; parts of the labels/descriptions for the ranges in these spreadsheets approximate categories of the 2nd kind; some of the fields in the existing actuarial database almost correspond to the categories of the 3rd kind; and groupings of items in the summary of results affect selection of the categories of the 4th kind.

Implementation Issues

All OO theory can be irrelevant if one cannot implement or emulate an actuarial data processing system as an Object-Oriented application. Fortunately, it is not only possible, but it has been already done: there exist several OO actuarial systems, including a few designed and implemented by the author.

Possible approaches to the design of such a system may include the following major tasks:

- *actuarial data arrays can be implemented as a hierarchy of abstract data types*
- *actuarial methods can be wrapped into functional classes*
- *persistence can be achieved by storing objects in an Object-Oriented (or Object-Relational or just plain Relational) Database*
- *links to Actuarial Data Mart can be added to import object's data and to export results of analysis ([9])*
- *a flexible user interface can be added to finalize construction of the OO actuarial system*

Classes in OO application may have different behavior and thus can be used for different purposes. Classes with the principal responsibility of maintaining data information are called abstract data types or data managers. Classes with the principal responsibility of assisting in the execution of complex tasks called functional classes or facilitators. The distinction between abstract data types and functional classes is somewhat similar to the distinction between nouns and verbs in a sentence.

An abstract data type is a logical extension of a programming language's built-in data types (integer, boolean, character) with a clear separation of the external interface and internal implementation. Abstract data type is a class dedicated to the representation of the complex data structures along with necessary additional functionality for storage, retrieval and transformation of the data. A good example of an abstract data type would be "Date": it does not matter how "Date" is stored in that class as long as users have an ability to request date to be displayed in any given format, retrieve year, month or day and perform date arithmetic.

Functional class is a natural extension of the programming language's built-in functions and operators. Packing several functions, associated with some kind of real-life activity, along with shared data, functional classes can be compiled into active components sometimes called engines. Good examples of functional classes would be the simulation engine of "@Risk", the optimization engine of "Solver" and the calculation engine of "Excel".

If triangles (rows, columns and diagonals) are essentially data manager classes, that is, abstract data types, then encapsulated actuarial algorithms (actuarial methods) are functional classes.

2nd rule of thumb. To decide which actuarial operation belongs to the data manager class (i.e., has to be implemented as a method in the abstract data type) vs. functional class, one should consider the following aspects of the algorithm:

- (a) whether or not the algorithm is subject to future modification (always the same "accumulation of the triangle" vs. always improving "calculation of the tail factor")
- (b) whether or not it is generic or specific for the particular data type ("summation of any two triangles" vs. "annualization of the inflation rate" applicable only to inflation vector)
- (c) whether or not it is user interruptible (automatic "extraction of the last diagonal" vs. "loss development method", which requires user selection)

In short, if an algorithm is a standard simple transformation of an object, it is most probably a method of the data class, and conversely, if an algorithm constitutes an actuarial method, it most probably belongs to the functional class.

Example 2. It makes a lot of sense to inherit Triangle, Vector and Diagonal actuarial classes from the Matrix class. Matrix implementation in existing spreadsheets or ActiveX components is extremely rich with properties). The OO designer just has to implement a few methods to create an algebra for triangles: the base transformations which would reduce operations on triangles to well-defined operations on matrices (inheritance at its best):

- *DiagonalsToColumns*
- *DiagonalsToRows*
- *RowsToDiagonals*
- *ColumnsToDiagonals*
- *DiagonalToVector (DiagonalNumber)*
- *VectorToDiagonal (DiagonalNumber)*
- *LastDiagonal*

For example, applying calendar year inflation to the triangle can be performed as a triad:

```
Triangle.DiagonalsToRows <-  
Matrix.MultiplyByVector (InflationVector) <-  
Triangle.RowsToDiags.
```

in more conventional notation:

```
Triangle.DiagonalsToRows.MultiplyByVector (InflationVector)  
.RowsToDiagonals
```

Or, even less intimidating, taking the average of the last 3 *diagonals* can be reduced to the average of the last 3 *rows* in the matrix if `Triangle.DiagonalsToRows` is implemented (see Figure 7.)

Note that because `Triangle` inherits from the `Matrix`, it can use operations available to `Matrix`, in particular, multiplication by vectors and taking the average of its rows. Most of actuarial algorithms can be expressed through a very limited set of basic triangle and matrix operations; for the rest of algorithms users always have access to matrix elements.

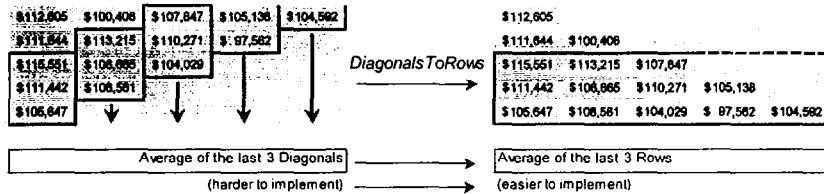


Figure 7

Example 3. A simple Chain-Ladder method rewritten in an OO fashion.

- `Step1 = InputTriangle.Accumulate(fromFirst, byAddition);`
- `Step2 = Step1.Shift(toLeft, by:= 1)/Step1;`
- `Step3 = Step2.DiagonalsToRows.RowsAverage(SelectedAverage);`
- `Step4 = UserSelectedVector(Step3);`
- `Step5 = Step4.Accumulate(fromLast, byMultiplication);`
- `Step6 = Step1.LastDiagonal(asColumn) * Step5.Invert;`

Or even shorter (assuming the `Input Triangle` is cumulative):

```
EstimateOfUltimate = InputTriangle.LastDiagonal(asColumn) *
UserSelectedFactors(default:=
InputTriangle.AgeToAgeFactors.Average(Medial, 5))
```

A complete actuarial system has to extend its classification to include objects used by all types of actuarial activities: reserving, pricing and finances. Policy objects – highly structured entities which store several dates along with the list of coverages and vectors of limits and attachment points – can be arranged in a hierarchy of their own (in such a hierarchy, finite reinsurance policy class can be derived from quote-share treaty class by adding aggregate limits property.) Vectors

of inflation rates, sets of statistical distribution parameters and a simulation engine – these are the primary examples of actuarial objects to be included in the system.

An important implementation consideration is the links to the actuarial Data Mart or equivalent source of actuarial data. The structure of that data depository may impose restrictions (and requirements) on the availability of some desired categories and members in a hierarchy, and, therefore, the structure of the existing Data Mart should be a very important consideration in OO Analysis. It would be wise to build into the system an ability to anticipate future changes in the structure of available actuarial data and adapt its hierarchical organization to it, in other words, to build support for dynamic (data driven) hierarchy.

Currently pure OO databases and languages are not as ubiquitous as their relational and functional counterparts: Oracle, SQL Server and Sybase (the most ubiquitous databases) do not support inheritance and to call Visual Basic (one of the most ubiquitous programming environments) Object-Oriented is a very big stretch. Nevertheless, these impure OO environments support enough OO features for building applications and systems based on the main OO principles. In instances, when particular OO feature is not natively supported, it usually can be effectively emulated, so users and designers can reap all the benefits of OO applications today. In fact, a pure OO implementation of the actuarial system is less important than thorough and systematic OO Analysis of the actuarial workflow; that is, rethinking the whole actuarial process in terms of objects, methods, hierarchies and classifications.

Note how important the selection of a proper hierarchy is: we started discussing actuarial data chunks' categories and suddenly all the industry buzzwords like "Data Mart," "Object-Oriented Analysis and Design," "client-server architecture" and "data-driven technologies" came into play.

With the advent of OO databases, which store objects and thus have to store data along with operations, there are even more places for execution of the programming code. Indeed, where to implement object's functionality: on the server or on the client, inside the database or outside? Standard transformation routines, which are not subject to frequent modifications and user interruption, that is, abstract data types methods, are better placed on the server. Indeed, why request a triangle and then accumulate it on the client – let the powerful server accumulate it and transfer the result; or why request the whole triangle when only last diagonal is needed – let the server extract it before transferring the result. As for functional classes (actuarial methods) they also may take advantage of the server through request brokers like CORBA or DCOM. So, the system designer can build a distributed multi-user application using these tested and optimized actuarial procedures (implemented as methods of the functional classes) as construction blocks.

The author does not believe in a single monolithic application simultaneously suitable for pricing, reserving and financial analysis - he rather prefers a suite of applications each highly optimized for particular purpose, but founded on a base of comprehensive yet coherent set of common components (classes). Proper design and classification of actuarial objects, both abstract data types and functional classes, will enable actuaries to build such applications themselves.

Conclusion

Inheritance, a necessary requirement for any Object-Oriented system, naturally generates an internal hierarchy of the actuarial objects, while the database's requirement for identity of every object imposes an external hierarchy on the actuarial objects. This duality of the hierarchy reflects the fact that some categories in classification are used to determine which actuarial algorithm to use and represent differences in an object's internal structure and behavior, while other categories exist only to distinguish similar objects and define groups for aggregations. In other words, the external hierarchy is just a factorization of all actuarial objects by internal hierarchy. A deep understanding of these two distinct sources of hierarchies helps to optimize categorization of actuarial objects for their intended use – actuarial analysis, and also provides a basis for more effective and robust Object-Oriented actuarial applications.

Acknowledgements

The author would like to thank all the members of his immediate (“internal”) and extended (“external”) family for their understanding and support during the long hours dedicated to the work on this paper and taken away from them.

Stamford, 1998

Appendix

Code samples are for illustration purposes only.

Example 1. The following text is a fragment of "LinearStorage" class implementation. Placed in

```
<class LinearStorage>

Private DynaStore() As Variant
Private nRows As Integer
Private nCols As Integer
Private nSize As Integer

Public Sub StoreTriangle(ByRef InputTrig As Variant)
    Dim i As Integer
    Dim j As Integer
    Dim k As Integer

    nRows = UBound(InputTrig, 1) - LBound(InputTrig, 1) + 1
    nCols = UBound(InputTrig, 2) - LBound(InputTrig, 2) + 1
    nSize = (1 + nCols - 2) * (1 + nCols - 1) / 2 + nRows

    ReDim DynaStore(1 To nSize)

    For j = 1 To nCols
        For i = 1 To nCols - j + 1
            k = (i + j - 2) * (i + j - 1) / 2 + i
            DynaStore(k) = InputTrig(i, j)
        Next i
    Next j
End Sub

Public Function RetrieveTriangle() As Variant
    Dim i As Integer
    Dim j As Integer
    Dim k As Integer
    Dim Output() As String

    ReDim Output(1 To nRows, 1 To nCols)

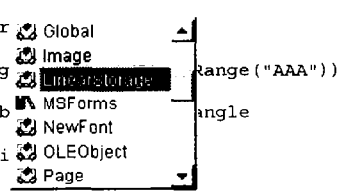
    For j = 1 To nCols
        For i = 1 To nCols - j + 1
            k = (i + j - 2) * (i + j - 1) / 2 + i
            Output(i, j) = DynaStore(k)
        Next i
    Next j

    ShowTriangle = Output
End Function
```

the VBA class module, this code will define an abstract data type called LinearStorage that will immediately become available along with VBA built-in data types.

```
Option Explicit

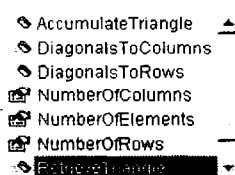
Function test() As Variant
    ' ...
    Dim TrigAsObject As New
    Dim TrigAs2DArray As Var
    TrigAsObject.StoreTriang
    ' ...
    TrigAs2DArray = TrigAsOb
    ' ...
    Set TrigAsObject = Nothi
    ' ...
End Function
```



It's *public* functions ("methods") and subroutines ("properties") will be available to all instances of this class.

```
Option Explicit

Function test() As Variant
    ' ...
    Dim TrigAsObject As New LinearStorage
    Dim TrigAs2DArray As Variant
    TrigAsObject.StoreTriangle (ActiveSheet.Range("AAA"))
    ' ...
    TrigAs2DArray = TrigAsObject.
    ' ...
    Set TrigAsObject = Nothing
    ' ...
End Function
```



Encapsulation. Note, that LinearStorage class includes both data (*nRows*, *nCols*, *nSize*, etc..) and operations (*StoreTriangle*, *RetrieveTriangle*, etc..). External programs will not have direct access to any variables we store in the class as well as to any operations we designate as *Private*: When the designer wants external programs to access some data (e.g., *nRows*) he will implement a dedicated operation (e.g., *NumberOfRows*), where class will have a chance to validate input and perform necessary transformations of related items (e.g., *nSize*.)

Glossary

A list of the most popular and influential variants of definitions for the most important OO concepts (mostly from [6] and [10]). Items are listed in the order of appearance in this article.

object *n.* 5.(a) any instance of one or more classes or types... 2.(b) any encapsulation of properties (e.g., data) and behavior (e.g., operations)... 1.(c) any real or abstract thing about which we store data and the operations to manipulate those data... 2.(a) any identifiable, encapsulated entity that provides one or more services that can be requested by a client... 1.(a) any abstraction that models a *single* thing... 9. any person, place or thing...

Synonym: INSTANCE

class *n.* 5. any set of objects that share the same or similar features... 4.(b) any implementation of a type of objects, all of the same kind... 2. any possibly generic factory of instantiation of instances... 7. the unit of modulation, data hiding, and encapsulation... 1.(b) any concept that has members... 1.(a) any uniquely-identified abstraction (i.e., model) of a set of logically-related instances that share the same or similar characteristics...

Synonym: TYPE

encapsulation *n.* 1.(b) the packaging of operations and data together into an object type such that the data are only accessible through messages to the object... 1.(a) the physical localization of features (e.g., properties, behaviors) into a single black-box abstraction that hides their implementation behind a public interface...

Synonym: INFORMATION (DATA) HIDING

inheritance *n.* 1.(b) the construction of a definition by incremental modification of other definitions... 3.(b) a mechanism that permits classes to share characteristics...

polymorphism *n.* 2. the ability of a single name to refer to different objects (i.e., objects of different classes)... 1.the ability of a single name to refer to different things having different forms...

hierarchy *n.* 1. any ranking or ordering of abstractions into a tree-like structure...

object identifier (OID) *n.* 1. the simple identifier permanently assigned to each object that is a) unique within some scope (i.g., an application), b) independent of the object's properties and state, c) constant during the existence of the object...

identity *n.* 1. the use of identifiers rather than keys[†] to uniquely identify objects...

[†] keys (fields) is a notion from the Relational Database vocabulary

persistence *n.* 1. the ability of an object to continue to exist after the execution of the program, process, or thread that created it...

object-oriented programming *n.* 1. any application specific programming resulting in programs that consist of collection of collaborating objects, which have a unique identity, encapsulate properties and operations, communicate via message passing, and are instances of classes related by inheritance, polymorphism and dynamic (run-time) binding...

dimension *n.* 2. an index for identifying values within a multi-dimensional array... 1. A *dimension* is a structural attribute of a multi-dimensional array that is a list of members, all of which are of a similar type in the user's perception of the data.

Example: months, quarters, years, etc., make up a time dimension; cities, regions, countries, etc., make up a geography dimension.

dimension member *n.* 1. a discrete name or identifier used to identify a data item's position and description within a dimension...

member combination *n.* 1. an exact description of a unique cell in a multi-dimensional array, consisting of a specific member selection in each dimension of the array...

generation *n.* 2. in a hierarchy, the distance from the top... 1. members of a hierarchy have the same generation if they have the same number of ancestors leading to the top...

Example: in a time dimension years are generation 1, quarters are generation 2, etc..

level *n.* 2. in a hierarchy, the distance from the bottom... 1. members of a dimension with hierarchies are at the same level if, within their hierarchy, they have the same maximum number of descendants in any single path below...

Example: in a time dimension months are level 0, quarters are level 1, etc..

Bibliography

- [1] David Brown. *An Introduction to Object-Oriented Analysis. Objects in Plain English.* 1997, Wiley
- [2] James Martin, James J. Odell. *OO Methods.* 1998, Prentice Hall
- [3] Mary E.S. Loomis. *Object-Oriented Databases: The Essentials.* 1995, Addison-Wesley
- [4] Douglas K. Barry. *The Object Database Handbook.* 1996, John Wiley&Sons
- [5] Michael Bhala, William Premerlani. *Object-Oriented Modeling and Design for Database Applications.* 1998, Prentice Hall
- [6] Donald G. Firesmith, Edward M. Eykholt. *Dictionary of Object Technology. The Definitive Desk Reference.* 1995, SIGS Books
- [7] Murray R. Cantor. *Object-Oriented Project Management with UML.* 1998, Wiley
- [8] Grady Booch. *Object-Oriented Analysis and Design with applications.* 1994, Addison-Wesley
- [9] Aleksey S. Popelyukhin. *The Big Picture: Actuarial Process from the Data Processing Point of View.* 1996, Library of Congress
- [10] *OLAP and OLAP Server Definitions.* 1997, OLAP Council

