Journal of International Information Management

Volume 5 | Issue 2 Article 7

1996

The object-oriented paradigm as an implementation of systems theory in IS

William A. Newman
University of Nevada, Las Vegas

Anthony R. Hendrickson
University of Nevada, Las Vegas

The object-oriented paradigm as an implementation of systems theory in IS

William A Newman
Anthony R. Hendrickson
University of Nevada, Las Vegas

ABSTRACT

Classical General Systems Theory (GST) is rich with fundamental concepts relating to the explanation of how systems behave and operate. The new concept of object-oriented techniques and methodologies was found to be closely aligned with most of the fundamental theories of GST. This article presents these alignments and similarities and clearly shows that object-oriented techniques provide a better fit to GST than older design methodologies.

INTRODUCTION

Classical General Systems Theory (GST) as we know it was derived from theorists working from approximately 1920 to the 1950s. These theorists first attempted to understand the behavior of organic systems by formulating a set of general system frameworks. Systems Theory has expanded over the years to now include inorganic systems and identifies fundamental principles about the behavior of these systems and their makeup. Applying the concepts of Systems Theory to the object-oriented paradigm clearly shows that object-oriented concepts better follow the fundamental concepts system theory. This article briefly discusses both the concepts of system theory and object-oriented methodologies and expands earlier work[1] characterizing the relationship of classical systems theory with the practical world of Local Area Networks which showed distinct parallels in system theory and practical computing methodologies in form and function.

HISTORY OF SYSTEMS THEORY

A formal theory of the behavior and fundamental principles surrounding all systems was derived from a variety of disciplines around the 1950s and numerous books and papers have been produced since then, further expanding the concepts and intricacies of systems. General Systems Theory itself evolved from the biological sciences and attempted to explain the behavior of organic systems after biologists became disenchanted with the analytical approach. The analytical approach attempted to understand systems by just examining their components, ignoring the relationships of the components to the larger entity. Fundamental GST works[2] like "Die Physischen Gestalten in Ruhe und im stationarem Zustand" laid the basis for looking at a system as an assemblage of parts making up a whole and included the Gestalten concept of holistic behavior - understanding the system by understanding its fundamental makeup and its relationship to other systems and objects within the system itself. Ludwig von Bertalanffy, a biologist, is considered the originator of GST[3] and formulated the concept that all living systems are open systems and interact with their environment.[4] Later, writers like Kenneth Boulding, Norbert Weiner, and Herbert Simon included inorganic systems in their fundamental system models and greatly expanded the boundaries of the discipline. Churchman[5] formulated five basic considerations that should be examined to understand the behavior and function of systems:

1. The objectives of the system and measurement instruments
2. The environment of the system
3. The resources of the system
4. The components of the system
5. The management of the system

Each of these subparts of a system must be examined in order to understand that system and constitute the holistic view. The system examination is incomplete if less than all five areas have been investigated.

WHAT IS A SYSTEM?

Numerous definitions exist for what a system is; however, all the definitions center on defining systems as being made up of objects or entities which have relationships among themselves. The entities have attributes that determine how these entities communicate and interrelate with themselves and other systems. Some of the definitions of a system are complex while others are simple. Two examples of a definition of systems are:

> A system is a device, procedure, or scheme which behaves according to some description, its function being to operate on information and/or energy and/or matter.[6]
>
> A system is a complex of interrelated entities.[7]

For this discussion, we define a system as an assemblage of components designed for some purpose to achieve an objective. These components, objects, or entities always share the following fundamental implications:

1. A system must be designed to accomplish an objective.
2. The elements of a system must have an established arrangement.
3. Interrelationships must exist among the individual elements of a system, and these interrelationships must be synergistic in nature.
4. The basic ingredients of a process (the flows of information, energy, and materials) are more vital than the basic elements of a system.
5. Organization objectives are more important than the objectives of its elements, and thus, there is a de-emphasis of the parochial objectives of the elements of a system.[8]

To paraphrase the above, all systems have organization, interaction, interdependence, integration and purpose. Systems have a hierarchy within other systems and among their components. Systems also have attributes or properties that manifest both themselves and their relationships to other objects and systems. Systems live within a definable boundary that differentiates them from their general environment. Systems perform on the basis of some standards set for the system and systems are controlled by at least one controller or organizer whose job it is to see that the system follows the standards that have been set. Systems communicate with the other objects in the system through outputs and inputs and with the external world or other systems through feedback. Feedback can either encourage correct behavior of the system (positive feedback) or bring the system back into conformity with its objective (negative feedback). Without feedback and maintenance, systems slide inevitably toward collapse (entropy). Negative entropy maintains the system and may in fact improve it.

HISTORY OF OBJECT-ORIENTATION

The concepts of code modularity, loose coupling, high cohesion, and information hiding, which are most closely tied to object-orientation were proposed by a number of software developers beginning in the early 1960s. Simula I and Smalltalk, both object-based languages, were conceived and initially appeared in the 1960s.[9] Several early attempts at incorporating these concepts into the structure of a language were made by various researchers. Yonezawa and Torkoro state:

> The term 'object' emerged almost independently in various fields in computer science, almost simultaneously in the early 1970s, to refer to notions that were different in their appearance, yet mutually related. All of these notions were invented to manage the complexity of software systems in such a way that objects represented components of a modular decomposed system or modular units of knowledge representation.[10]

However, no one person or group can be credited with inventing the paradigm. Advances in computer architecture, programming languages, and design methodologies all contributed to the development of OO concepts.[11] The object concept began in hardware technology with descriptor-based and capability-based designs in the 1970s.[12] These designs attempted to bridge the gap between low-level machine languages and higher level abstractions necessary in modern programming languages.[13]

Smalltalk (versions 72, 74, 76, and 80) built upon Simula's OO paradigm and conceptualized everything into an object which belonged to some object-type class. Other languages such as Alphard, CLU, Euclid, Gypsy, Mesa, and Modula all attempted to extend the ideas of data abstraction and build higher level programming languages.

Dijkstra[14] first introduced the concept of using layers of abstraction to create systems comprised of component modular design. Dijkstra's THE language was the culmination of this layered approach to OO architectures. Independent of programming languages, database technology evolved primarily via the entity relationship (ER) concept of Chen.[15] The ER model conceptualizes entities with attributes and relationships between the entities. Rumbaugh[16] proposed an integration of the ER model with an OO approach.

Along the way other researchers have contributed portions that now comprise the OO paradigm. Parnas[17] introduced the notion that aspects of data do not always need to be visible to the user of the data. This concept of information hiding restricts the details of how an operation is accomplished. While the operation is known, the actual process used to complete the operation is not. This is necessary to protect the data from arbitrary and/or unauthorized use. Mechanisms for abstract data typing were developed by several researchers in the 1970s.[18] Finally, an underlying theory of abstract typing and subclasses was presented by Hoare.[19]

Although OO concepts were initialized some 20 to 30 years ago, they have not gained widespread industry popularity and acceptance until recently. Limitations related to computer architecture, programming languages, design methodology, database theory, and cognitive science have all contributed to the slow acceptance of OO concepts. However, OO concepts are closely aligned to the fundamental concepts found in systems theory and the OO paradigm may be seen as an implementation methodology and tool for systems theory concepts in IS. The next section will show the alignment that the object-oriented paradigm has with the GST model.

SYSTEMS THEORY AND OBJECT-ORIENTATION

Systems theory provides a natural way of conceptualizing and analyzing the real world within the information systems (IS) discipline. One problem with mapping system theory to IS however, is that the technology in past IS applications was relatively primitive (i.e., the model was not a good fit due to technological limits of the application software). Systems analysis and design techniques and programming languages of the past could not support the same level of abstraction embodied in systems theory. Thus, applications were record and file based, not entity or object-based.

OO concepts are closely aligned to the fundamental concepts found in systems theory and the OO paradigm may be seen as an implementation methodology and tool for systems theory concepts in IS. One of the fundamental problems with explaining the concept of object-orientation and its mapping to GST, is the lack of common terminology outside the concept on which to base the definition of the paradigm's terms. Each of the OO paradigm's terms seems to be currently explained with terms contained within the paradigm itself. The interdependence of the concepts in OO, like the interdependence of systems theory components, may contribute to the difficulty in easily understanding the approach. Goldberg and Robson summarize the problem in discussing the Smalltalk language:

> Smalltalk is based on a small number of concepts, but defined by usual terminology. Due to the uniformity with which the object-message orientation is carried out in the system, there are very few new programming concepts to learn in order to understand Smalltalk. On the one hand, this means that the reader can be told all the concepts quickly and then explore the various ways in which these concepts are applied in the system. These concepts are presented by defining the five words mentioned earlier that make up the vocabulary of Smalltalk - object, message, class, instance and method. These five words are defined in terms of each other, so it is almost as though the reader must know everything before knowing anything.[20]

OBJECTS

The term Object, is a key element of both the GST and OO concepts. An object in GST is often called an entity; however the terms are defined synonymously and refer to a fundamental building block of which the system is comprised. Objects under GST can range from the most simple (a grain of sand) to extremely complex (the universe). For one to understand a system, one must understand the objects or components from which it is made and how these objects behave.

In OO, the notion of the concept of objects lay within earlier work on Semantic Models dating back as far as Chen's Entity Relationship model[21] and many of the earliest investigations into this area refer to objects as entities, using the terms synonymously and seeking to define entities as objects in the real world. As the idea of object-oriented was conceived, however, the opposite became true and objects were defined as unique entities - real world occurrences of unique, self-contained items.[22] This is not surprising since the basis of much of this research is to accurately pattern systems development activities to the behavior of real world data elements.

Objects can be extremely simple, such as a single biological cell, or as complex as a complete anatomical life form. The level of complexity depends upon the level of abstraction one desires. At some level of abstraction the single cell would appear quite elementary. However, when one considers the complex operations performed by the cell to sustain life, then the object appears much more complex. This ability to modify our level of abstraction to fit the problem environment allows us to break extremely complex problems into manageable solutions. Thus, as Figure 1 shows, our level of abstraction creates simple and complex objects.

Figure 1. System Complexity

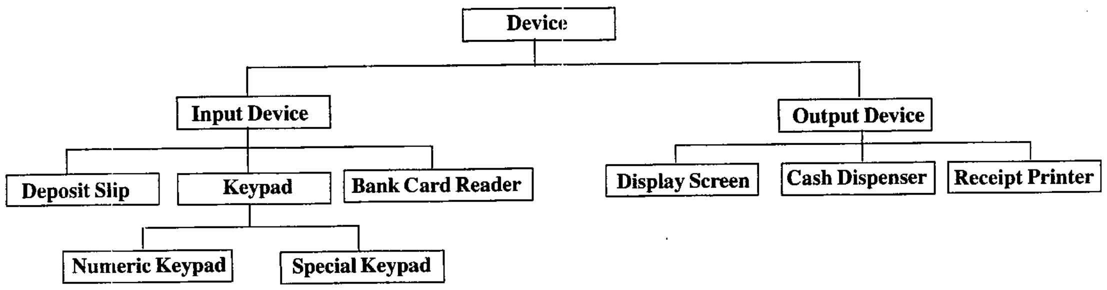
(figure 1 goes here)

CLASS

After objects, class is the next major construct of both the GST and OO paradigm. In GST, class is represented as the fundamental hierarchy of systems. Objects in the same frame make up a set. Aristotle was one of the first scientists to be credited with the classifying of the approximately, then known one thousand plants and animals. His work simplified the division of animals into animals with backbones and red blood and those without backbones and no red blood. He also classified plants by size and appearance. Later in the 18th century, Linnaeus reclassified plants by structured arrangement.[23] In 1971, Boulding represented his hierarchy of major system levels by modeling a structure of system complexity based on nine levels.[24] Again, the level of abstraction determines the level of complexity. Systems theory provides an abstraction tool for decomposing and aggregating object into manageable components.

In OO, paradigm objects are also organized hierarchically into classes. Objects that are of a similar type or kind are combined into common types. Since any object will contain certain characteristics which make it unique, it will also contain characteristics which will be shared in common with other objects. Then objects possessing the same or similar kinds of characteristics are grouped together in common groups called classes. A class is a logical grouping of objects based on criteria parameters of the characteristics possessed in common by the objects as a group. An object contains characteristics which provide for the object's unique distinction from other objects, and characteristics which help to identify the object as a member of a group of objects with the same or similar characteristics. The common characteristics become the basis for the objects to be classified.[25] Figure 2 shows the relationship of class to hierarchy.

Figure 2. The Hierarchy of Classes



(SOURCE: Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice Hall, p. 58.)

Within GST systems are decomposed into subsystems and aggregated into supersystems. More primitive systems can be combined to create higher order systems. The more primitive systems become subsystems of the higher order systems. Schoderbek, Schoderbek, and Kefalas state:

> While an obvious hierarchy of systems exists (the ultimate system being the universe), still, almost any system can be divided and subdivided into subsystems and subsubsystems depending upon the particular resolution level desired.[26]

Depending upon one's level of abstraction, the higher order system may be considered a supersystem of the component subsystems. Schoderbek, Schoderbek, and Kefalas further offer five propositions for systems hierarchy:

1. A system is always made up of other systems.
2. Given a certain system, another system can always be found that comprises it, except for the Universal System, which comprises all others.
3. Given two systems, the one system comprising the other can be called the high-level system in relation to the system it comprises, which is called the low-level system.
4. A hierarchy of systems exists whereby lower-level systems are comprised into high-level systems.
5. The low-level systems are in turn made up of other systems and can, therefore, be considered the high-level system for the lower-level systems to be found in it.[27]

A similar mechanism is utilized with the OO paradigm. Classes may be grouped into higher classes called *superclasses* or divided into subsets called *subclasses*. Classes, superclasses, and subclasses are the vehicle for grouping and subdividing objects into logical categories based on the instance variables, which are the criteria for the separation and/or accumulation of objects. In some definitions an object can belong only to one class, while in others, objects can be a member of multiple classes.

The real power of the OO approach lies within the ability to define instance variables which create new classes from existing objects. Therefore, more primitive objects can be grouped together to form more complex objects. Conversely, complex objects can be divided into subclasses creating more primitive objects.[28]

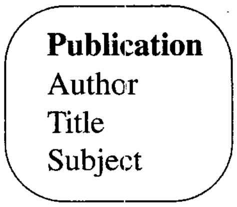
IS-A RELATIONSHIPS

The relationships between objects and classes are described generally as *IS-A* relationships. IS-A relationships form the hierarchical structure between objects and between classes. Since this structure is hierarchical in nature, it infers a strict series of one-to-many relationships. Any member of a subordinate class *is a* member of the superior class. As with the hierarchical models of database modeling, some researchers allow a parent node to have many children, but a child must have one unique parent. Others in the literature allow what is analogous to a network structure supporting many-to-many relationships. The IS-A relationship is maintained in that subordinate classes are members of superior classes; however, a child can be a member of many superior classes.[29]

AGGREGATION/GENERALIZATION

There are two major types of IS-A relationships, *aggregation* and *generalization.* Both of these concepts refer to a process of combining lower level objects and classes into higher level objects or classes. The distinction is the type of subclass or subordinate object that is combined into the higher level object or class. Aggregation is the combining of component pieces into the product entity. For example, the components of a computer would be aggregated to form the object *COMPUTER*. Also, the class of each component could be aggregated to form the class of *COMPUTERS.* Another example would be the aggregation of the components *BOOK TITLE, BOOK AUTHOR*, etc. into the higher product object *PUBLICATION.* [30] The inverse of aggregation is decomposition, the breaking apart of a higher object into its component parts.[31]

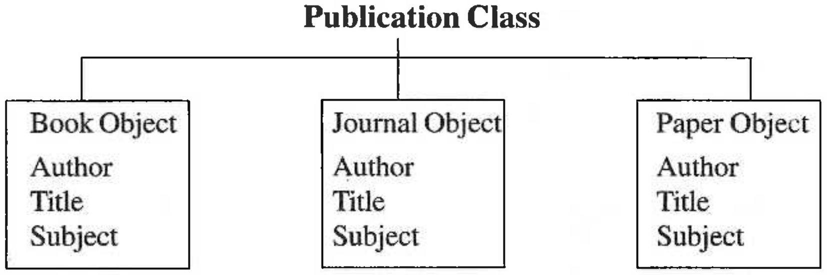
Figure 3. Aggregation



Aggregation of Component Objects

Generalization is the combining of lower level objects or classes into higher level classes by generalizing the attributes of the lower level components. In contrasting generalization with the *PUBLICATION* example in the previous paragraph, generalization would be the combining of objects or classes like *BOOK, JOURNAL PAPER,* and *CONFERENCE PAPER* into a higher level object or class called *PUBLICATION.* [32] Conversely, the combination of *COMPUTER* with *CALCULATOR, DESK,* and *FILE CABINET* would be a generalization into the class *OFFICE EQUIPMENT.* [33] *Specialization* is the inverse process of generalization. Specialization is the segmentation of a more general class into more specialized classes, i.e., *OFFICE EQUIPMENT* into *COMPUTERS, CALCULATORS*, etc.[34]

Figure 4. Generalization



Generalization of Objects into a Class

PROPERTIES/ATTRIBUTES/INHERITANCE

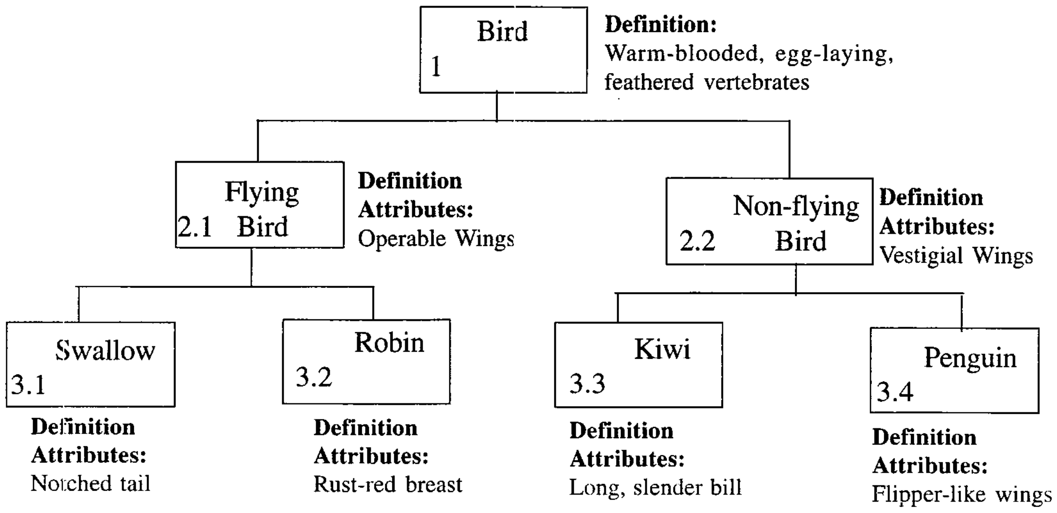
In GST, attributes are properties of both objects and the relationships of objects among themselves.[35] Attributes can take two forms. Deterministic attributes define the object itself while accompanying attributes are simply passed along as part of the object class. In OO, objects convey characteristic properties between lower level objects via inheritance. The inheritance concept is actually the possession of one or more accompanying attributes.

Inheritance is the concept that properties of superior objects or classes are passed along to subordinate objects or classes. Just as human characteristics are passed genetically from generation to generation, the characteristics of parent nodes of an object or class are passed to the child node. The type of IS-A relationship, whether aggregation or generalization, can provide clues to the type of attributes to be inherited by a child node. From the previous example of *PUBLICATION,* one can see the object *PUBLICATION* which is comprised of *AUTHOR* and *BOOK TITLE,* through aggregation, passes different attributes to the subordinate objects than does the object *PUBLICATION,* through generalization with *JOURNAL PAPER, BOOK,* and *CONFERENCE PAPER.*

Inheritance provides a means by which more complex objects and classes can be created. In terms of modeling, inheritance allows lower level objects or classes to automatically include aspects of their parent nodes. For example, a genealogical hierarchy for "Bird" shown in Fig. 5. This structure illustrates how subclasses and their related objects inherit attributes from their respective superclasses. A general class is created which contains the defined attributes of all subordinate objects. The definition then carries through to the subordinate classes and objects.

Additional characteristics attributed to only a subclass can be added at the subclass level. In the illustration in Figure 5, the attributes of "warm-blooded," "egg-laying," and "feathered vertebrates" hold true for all subordinate objects, regardless of their subclassification. The attribute of "vestigial wings" in the subclass *NON-FLYING BIRD,* is inherited only by the members of the subclass (*KIWI* and *PENGUIN*). In each case, the attributes do not need repeating within the subordinate object or subclass. Simply arranging the objects in the hierarchical structure shown, attributes the corresponding attributes to the class or subclass via the inheritance concept. Thus, as subclasses such as *FLYING BIRD* and *NON-FLYING BIRD* are created there is no need to include the basic attributes in each subclass since they will be inherited from the superclass *BIRD.* [36]

Figure 5. Object Inheritance



(Adapted from: Henderson-Sellers and Edwards, *op cit.,* Ref. 22, p. 149)

Inheritance provides an implementation mechanism for the hierarchical structure. The hierarchical structure found in GST assumes that objects or entities farther down on the hierarchical structure logically are a subcategory of their corresponding parent object(s). The arrangement of entities in this manner on the hierarchy diagram conveys this concept, intuitively. However, the OO paradigm seeks to implement this concept in reality, not just in mental abstraction. Thus, inheritance provides a mechanism for implementing this intuitive concept into a workable operation.

Inheritance is a major component in the reusability of code discussed later. Since objects can assume attributes of their parent nodes it is not necessary to code the same variable values for each object of a class. Instead, the attributes of the parent class are automatically incorporated into each object instance, simply by forming a class which contains the attributes required by each object instance. Thus, from a programming standpoint, the concepts of class and inheritance assist in making code modular and therefore more reusable.[37]

Inheritance provides for extensibility in the modeling process. That is, the models of OO approaches can be extended or built upon without adversely affecting the implementation of the work already accomplished. Through inheritance this is accomplished because each subsequent class or object acquires attributes from its predecessor(s).[38] When a strict hierarchical structure is observed when creating an IS-A relationship the inheritance is referred to as single inheritance. This is because the subordinate object or class has only one parent. Therefore, attributes can be acquired from one source. However, when there is no strict adherence to the hierarchical structure, a subordinate object or class can acquire attributes from multiple parents, hence multiple inheritance.[39]

An object's characteristics is commonly referred to in the literature as *attributes.* A class then is defined as a logical grouping of objects based on criteria parameters of the attributes for that particular class. These criteria parameters are called *instance variables.* A single occurrence of an object in a class is an instance. The variables that describe that particular group of objects are the instance variables. For example, the object *STUDENT* may be grouped into classes based on academic major. Academic major then is one of the instance variables for class *ACCOUNTING-STUDENTS.*

Some in the literature prefer the terminology *instance variable,*[40] while others prefer the term *attribute,*[41] and still others use the terms synonymously within their work.[42] With either term the concept remains to distinguish objects from one another and to augment objects with common characteristics.

ENVIRONMENT/ABSTRACTION

According to the GST, a system operates within a known environment. The system environment is comprised of system inputs, system processes, and system outputs. While the system itself is limited to items directly in the system's span of control, the system environment includes those things that are input into the system which determine how the system behaves and those things that the system produces, its outputs. Everything within the complete control of the system is part of the system.

Everything outside the system's complete control is external, but not necessarily within the system environment. In order for something to be part of the system's environment, that thing must exert some significant force which affects the system's performance. If an external object has no significant impact upon the performance and operation of the system, then that object is outside the system environment. Thus, for an object to be part of the system it must be both relevant and controllable. If it is only relevant and not controllable, it is then part of the environment. Finally, if the item is neither relevant nor controllable, it is outside the system's environment.[43]

A key aspect of any system is the connection between system inputs and system outputs. Within any system, some mechanism exists which links the system outputs to the items input into the system. This connection is the feedback mechanism. This connection may be quite strong or virtually non-existent, depending upon the system. However, feedback is always present to some extent, even if it is merely a chain reaction between other entities within the system environment. The more direct the feedback controls, the greater the ability of the system to adjust to changes within its system environment.[44]

Closely related to any discussion of system environments is the concept of system boundaries. System boundaries are usually determined by an arbitrary delineation between the objects of interest and those that are irrelevant. The concepts included in a model of a real world system are determined based upon the observer's interest and determination of relevance.

The process of creating modeling representations of real world objects is commonly referred to as abstraction. This process of abstraction is a necessary tool for the creation of more complex operations and more complex relationships between data elements. One aspect of an object is that it contains abstract qualities about the real world entity. The object contains a degree of abstraction because it inherently carries qualities about the real world entity, which are meaningful to the user of the model. For example the object *STUDENT* would inherently contain information about the real world entity *STUDENT.* These qualities may include *NAME, ADDRESS, STUDENT-ID#,* etc. The user of the model would conjure a whole list of properties about the object *STUDENT* without any further prompting. The idea of creating objects is that this abstraction quality allows the user of the model to relate a host of ideas about the entity without listing each specific item. Thus, abstraction embodies the object with a concrete tie to reality.[45]

Obviously, this concept of abstraction is extremely necessary for successful modeling. The design and analysis functions are accomplished much more efficiently and effectively if the end-users can better understand the models of the process as laid out by the designers. This in turn is accomplished more effectively if the model itself reflects reality as much as possible.[46] Simple procedures and projects require this to a lesser extent than more complex ones. As the level of complexity of an application increases, the need for abstraction increases.[47]

This overall attempt to isolate the idea or concept of the data item from the implementation and/or manipulation of the data item is also part of the term abstraction. This aspect of abstraction is very closely analogous to independence in most database management contexts. The abstraction property is seeking to insulate the data element concept from its manipulation in an effort to allow more powerful modeling to be accomplished with the element.[48]

Traditional record-based approaches lack the abstraction capabilities to link real world items to elements in the most models. One of the major advantages of the OO approach is the superior abstraction capabilities of the approach over traditional record-based models.

Researchers have embraced the object concept because the abstraction capabilities prove to be a powerful tool for providing this abstract linking to real life items.[49] As discussed earlier this is a major advantage as the complexity of the application increases. This link to real world items is not a necessity for investigators of programming languages. Their focus is on the manipulation of data regardless of the data's representation to real life items. Instead, programming languages concentrate on defining objects as data elements in which the data, and the procedures which operate on the data, are contained within one entity. This act of combining the data and its procedures is referred to as *encapsulation.*[50]

ENCAPSULATION/EQUIFINALITY/POLYMORPHISM

Encapsulation is one of the most important aspects of programming in the OO paradigm. Encapsulation is the main form of implementing the abstraction principle within OO programming. The concept of abstraction requires that an object inherently contain the properties of real world entities. This is accomplished at least in part through encapsulation. The premise of the concept is to package together in one entity or object the data element and its corresponding attributes or properties.

In the programming task, this requires that data elements and their corresponding procedures are viewed, accessed, and manipulated in concert. Data elements and their specific operands grouped together into a 'capsule' of knowledge. This capsule of knowledge is the object. Thus, one part of the process of creating an object is the aspect of encapsulation.[51] When data is encapsulated with its operands, the user of the data element no longer is obliged (or able) to manipulate the data as they see fit. The user of the data element may request the manipulation of the data in a particular manner but the actual processing is transparent to the user. A user has visibility to the operations available, but not how these operations will actually be executed.[52] This concept is developed and explained further in a discussion of *messages* and *methods,* later in this article.

For now, it is important to understand that encapsulation is critical to the concept of object. For the object *STUDENT,* its properties such as *NAME, ADDRESS, STUDENT-ID#,* etc. would be contained within the object along with the procedures for manipulating these properties in order to provide information to users outside the context of the object itself. For a less complex object, *INTEGER,* there would be a few properties outlining its size parameters. The procedures appropriate for its manipulation such as addition, subtraction, multiplication, squaring, etc. would also be contained within the object.

In GST systems, it is recognized that entities or objects may possess multiple goals. Objects may seek to satisfy these multiple goals simultaneously. Additionally, GST provides that biological or social systems are not mechanistic and often achieve their goals utilizing different initial conditions and different methods. Within the OO paradigm, multiple objects continuously interact with one another. This dynamic interaction often requires dissimilar objects to behave in heterogeneous ways. OO allows this to occur via a concept called polymorphism.

Polymorphism allows objects to react differently to different stimuli. Depending upon the methods encapsulated in an object and the coded procedures embedded in it, an object may accomplish a task in different ways at different times. The combination of external stimuli and coded procedures allow objects to function dynamically. Additionally, objects not only react differently themselves, but different objects can behave differently when they encounter the same stimuli. This is obvious in natural systems. Chemicals, for example, react differently to alkaline and acid, based upon their own molecular structure. The OO paradigm attempts to more closely link the real world to software development. Polymorphism provides a mechanism for the implementation of multiple goal seeking and EQUIFINALITY within the OO paradigm.

MESSAGES/METHODS

In the traditional data processing approach, data elements are manipulated by the procedures coded by programmers for each particular application. A great deal of effort is usually expended to ensure the separation and independence between data elements and the procedures which act upon them. Since one of the major premises of the OO approach is that the procedures are encapsulated with the data elements themselves, a new methodology for processing the data is required.

The encapsulation of procedures with data elements creates an aspect of public and private access. Portions of the object's properties are available for access by all users who have access to the object itself. These properties are considered public. The operations available for use on an object are part of the object's interface with the rest of the world. The operations available are public; however, how the operations are accomplished are private. For example if an object *INTEGER* has a value of "6" and a procedure *PRODUCT* available, the operation of procedure *PRODUCT* is public. Users having access to the *INTEGER* "6" will know that *PRODUCT* is an operation available for use on "6." However, the process by which *PRODUCT* is accomplished is private. If a user were to request the *PRODUCT* of 3 times "6," the process may be accomplished by "3 x 6" or "6 + 6 + 6". How the task is accomplished is trivial. The user only wants the information "18," the *PRODUCT*, the output of the operation.

REUSABILITY/MAINTENANCE/ENTROPY

The second major concept of the OO paradigm is reusability. Reusability, inheritance, and encapsulation comprise the major components necessary for the implementation of the OO approach. The OO paradigm and GST have a hierarchical structure. Inheritance allows the systems farther down the hierarchical structure to logically inherit the characteristics and attributes of the higher order systems. The hierarchical structure of GST provides for complex systems to be constructed from more primitive subsystems. Just as inheritance provides an implementation mechanism for the transfer of object characteristics, reusability provides the OO paradigm with an implementation strategy for the construction of complex objects from simple objects. For example, in biological systems simple cells are aggregated to create increasingly complex systems and supersystems. Reusability provides this aggregation mechanism, within the OO paradigm.

Reusability implies a certain degree of modularity in code that is developed. The essence of the OO paradigm from the programming aspect is the idea of reusability of code. The only real benefit to programmers from using this approach is that it promotes a highly structured environment and ability to create more complex processing via previously defined, coded, and compiled code. While encapsulation is necessary in programming for the OO approach to be implemented, it is a very difficult task to encapsulate procedures with data elements. However, the reusability of code is a highly desirable aspect of the approach.[53]

While technically the OO approach could possibly be implemented without reusable code, there would be little, if any, tangible benefits from a programming standpoint of using the approach. Cox[54] stresses this issue of the paradigm by presenting an analogy between OO programming and electronic integrated circuits. He terms this "software-ICs." Just as electronic circuitry is modular with boards being designed for a specific task, Cox[55] indicates that OO programming creates software modules that can be plugged into each other. As the overall application changes and requires modifications, the "software-ICs" can be changed to provide the currently required circuitry.

Reusability also addresses the problems of system maintenance and system entropy. System maintenance activities are designed to extend a system's life by upgrading its functionality for dynamic systems environments. Reusable code extends the application software life cycle by forcing highly cohesive, loosely coupled code modules which can be more easily modified and reused providing greater flexibility in application design. Since the code is forced to be more cohesive and more loosely coupled, its inherent life expectancy is longer and therefore system entropy is reduced.

CONCLUSION

The GST is a cornerstone of scientific theory. It provides an instrument for the evaluation and construction of highly sophisticated concepts in virtually all areas of scientific endeavor. GST represents a framework for ordering and organizing human cognition concerning the aspects of the world around us. Contrary to the analytical approach, which seeks to understand the component parts of an item, the systems approach attempts to examine something from a more holistic point of view. An item of interest is viewed as a component part of a larger whole. The item is analyzed as a subsystem of the larger system, and at another level of abstraction this subsystem is decomposed into its own component parts.

Within information systems, the historical approaches to application software development have failed to capture the essence of GST. Structured approaches to software development lack the abstraction framework presented in the GST approach. Structured approaches more closely resemble the analytical approach in their attempt to fully understand a specific application, or application function, without examining the function in terms of a related system of functionality. Applications are analyzed from a very limited view of functionality. An applications functionality is dissected and analyzed, and only then are specific interfaces to that functionality developed.

The OO paradigm offers an approach to application development that more closely resembles the framework set forth in the GST. The OO approach decomposes complex systems into successively smaller subsystems and subsubsystems. The more primitive or elementary objects are aggregated into more complex objects. The objects are arranged in a hierarchical structure by generalizing objects into object-type classes.

The OO concepts of inheritance, encapsulation, and reusability provide a functional means of implementing the concepts set forth in the GST. The paradigm allows objects to be decomposed into more manageable units. These discreet units are completely encapsulated objects of information, including data and the required procedures necessary to manipulate the data. Encapsulation provides a clear delineation between the internal object attributes and functions and the external object environment. At the object level encapsulation provides an implementation of the system boundaries concept found in GST.

The hierarchical structure of GST is facilitated within OO via the concepts of inheritance and reusability. Objects are generalized into classes based upon their attributes and behavior. Attributes and characteristics attributable to an entire class need not be repeated in every class member. Instead these characteristics and behaviors may be attributed to the class and each member object can inherit these traits in much the same manner as biological inheritance operates. Thus, inheritance provides a methodology for creating logical hierarchies of objects and classes.

Finally, reusability provides the OO paradigm with a method of incorporating the GST concept of interchangeable subsystems. Software designs and code are bundled into discreet conceptual modules. These modules can be interchangeably linked to form diverse software applications. This modularity also facilitates the hierarchical structure in that the modules are completely self-contained and thus can be rearranged without significant impact on other subsystems.

Since the OO paradigm evolved over an extended period of time and contributions to the approach came from a variety of disciplines, it would be inaccurate to present the paradigm as an approach designed specifically to implement the GST. However, this article clearly presents alignments and similarities which indicate that OO techniques provide a closer fit to GST than older design methodologies. As such, the value of the OO paradigm as a tool for system development is significantly enhanced.

ENDNOTES

[1] Newman, William A. (1989). Systems Theory and Local Area Networks. *Journal of Systems Management,* February, 1989.

[2] Kramer, Nic. J. T. A. and Smit, Jacob de (1977). *Systems Thinking.* Martinus Nijhoff Social Sciences Division, p. 3.

[3]Bertalanffy, Ludwig von (1950). The Theory of Open Systems in Physics and Biology, *Science,* Vol. III.

[4] Schoderbek, Peter P., Schoderbek, Charles G., and Kefalas, Asterios, G. (1985). *Management Systems: Conceptual Considerations,* Business Publications, Inc., Plano, TX, 3rd ed., p. 34.

[5] Churchman, C. West (1968). *The Systems Approach.* New York: Delacorte Press.

[6] Elis, David O. and Ludwig, Fred J. (1977). *Systems Philosophy,* Prentice-Hall Inc., Englewood Cliffs, N.J., p. 7.

[7] Ackoff, R. L. (1962). *Scientific Method: Optimizing Applied Research Decisions*, John Wiley & Sons, New York, NY.

[8] Luchsinger, Vincent P. and Dock, V. Thomas (1988). General Systems Theory. In *Readings in Information Systems,* West Publishing, p. 3.

[9] Martin, James and Odell, James J. (1992). *Object-Oriented Analysis & Design,* Prentice Hall, Englewood Cliffs, NJ.

[10] Yonezawa, A. and Tokoro, M. (1987). Object-Oriented Concurrent Programming: An Introduction. In *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA, p. 2.

[11] Levy, H. (1984). *Capability-Based Computer Systems.* Digital Press, Bedford, MA, p. 13.

[12] Ramamoorthy, C. and Sheu, P. (1988). Object-Oriented Concurrent Programming. *IEEE Expert, 3* (No. 13), p. 14.

[13] Myers, G. (1982). *Advances in Computer Architecture,* 2nd ed. New York, NY: John Wiley and Sons, p. 58.

[14] Dijkstra, E. (1968). the Structure of "THE" Multiprogramming System. *Communications of the ACM, 11*(No. 5).

[15]Chen, P. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems, I*(No. 1).

[16] Rumbaugh, J. (1988). Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM, 31* (No. 4), p. 415.

[17] Parnas, D. (1979). On the Criteria to Be Used in Decomposing Systems into Modules. In *Classics in Software Engineering*. New York, NY: Yourdon Press.

[18] Liskov, B. and Zilles, S. (1977). An Introduction to Formal Specifications of Data Abstractions. *Current Trends in Programming Methodology: Software Specification and Design, 1*. Englewood Cliffs, NJ: Prentice Hall; Guttag, J. (1980) Abstract Data Types and the Development of the Simula Languages. In *Programming Language Design*. New York, NY: Compute Society Press; Shaw, Abstraction Techniques.

[19] Nygaard, K. and Dahl, O.J. (1981). The Development of the Simula Languages. In *History of Programming Languages*. New York, NY: Academic Press, Inc., p. 460.

[20] Goldberg, A. and Robson, D. (1983). *Smalltalk-80 The Language and its Implementation*. Reading, MA: Addison-Wesley, p. 12.

[21] *Op. cit.*, Ref. 15.

[22] Bailin, S. C. (1989). An Object-Oriented Requirements Specification Method. *Communications of the ACM, 32* (No. 5), pp. 608-623; Blaha, M. R., Premerlani, W. J., and Rumbaugh, J. E. (1988). Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM, 31* (No. 4), pp. 414-427; Hammer, M. and McLeod, D. (1981). Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems, 6*(No. 3), pp. 351-386; Henderson-Sellers, B. and Edwards, J. M. (1990). The Object-Oriented Systems Life-cycle. *Communications of the ACM, 33*, (No. 9), pp. 142-159; Kroenke, D. M. and Dolan, K. A. (1988). *Database Processing - Fundamentals * Design * Implementation*. 3rd ed., Chicago, IL: Science Research Associates, Inc.; Maier, D. (1984). Capturing More Meaning in Databases. *Journal of Management Information Systems, 1*(No. 1), pp. 33-49; McFadden, F. R. and Hoffer, J. A. (1991). *Database Management*. 3rd ed., Redwood City, CA: Benjamin/Cummings Publishing Co., Inc.; Peckham, J. and Maryanski, F. (1988). Semantic Data Models. *Communications of the ACM, 20* (No. 3), pp. 153-189.

[23]*Op. Cit.*, Ref. 4, p. 41.

[24] Boulding, Kenneth (1971). General Systems Theory-The Skeleton of Science. In *Management Systems*, John Wiley and Sons. 2nd ed., p. 20-28.

[25] Agha, G. (1990). Concurrent Object-Oriented Programming. *Communications of the ACM, 33* (no. 9), pp. 125-141; Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., and Pintado, X. (1990). Class Management for Software Communities. *Communications of the ACM, 33* (No. 9), p. 90-103; Hammer and McLeod, *op cit.*, Ref. 22; Howard, G. S. (1988). Object-Oriented Programming Explained. *Journal of Systems Management*, July 1989, pp. 13-19; Korson, T. and McGregor, J.D. (1990). Understanding Object-Oriented; A Unifying Paradigm. *Communications of the ACM, 33* (No. 9), pp. 40-60; Lerner, B. S. and Habermann, A. N. (1990). Beyond Schema Evolution to Database Reorganization. In *Proceedings of OOPSLA '90 SIGPLAN Notices, 25* (No. 10), 67-88; Sciore, E. (1989). Object Specialization. *ACM Transactions on Information Systems, 7* (No. 2), pp. 103-000; Unland, R. and Schlageter, G. (1989). An Object-Oriented Programming Environment for Advanced Database Applications. *Journal of Object-Oriented Programming*. May-June 1989, pp. 7-19.

[26] *Op. cit.*, Ref. 4, p. 51.

[27] *Ibid.*

[28] Agha, *op. cit.*, Ref. 25; Banerjee, J., Chou, H., Garza, J. F., Kim, W., Woelk, D., Ballou, N., Kim, M., and Kim, H. (1987). Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems, 5* (No. 1), pp. 3-26; Cox, B. J. (1986). *Object-Oriented Programming, An Evolutionary Approach.* Reading, MA: Addison-Wesley Publishing Company; Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A., and Shan, M. C. (1987). Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems, 5* (no. 1), pp. 48-69; Gibbs, et. al., *op. cit.*, Ref. 25; Hammer and McLeod, *op. cit.*, Ref. 22; Hudson, S. E. and King, R. (1989). Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System, *ACM Transactions on Database Systems, 14* (No. 3), pp. 291-321; Lerner and Habermann, *op. cit.*, Ref. 25; Sciore, *op. cit.*, Ref. 25; Unland and Schlageter, *op. cit.*, Ref. 25; Wirfs-Brock, R. J. and Wilkerson, B. (1989). Object-Oriented Design: A Responsibility-Driven Approach. In *Proceedings OOPSLA '89 SIGPLAN Notices, 24* (No. 10), pp. 71-75.

[29] Banerjee, et al., *op. cit.*, Ref. 28; Blaha, et al., *op cit.*, Ref. 22; Hull, R. and King, R. (1987). Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys, 19* (No. 3), pp. 201-260; Papazoglou, M. P. and Marinos, L. (1990). An Object-Oriented Approach to Distributed Data Management. *Journal of Systems Software.* November 1990, pp. 95-109; Peckham and Maryanski, *op. cit.*, Ref. 22; Sciore, *op. cit.*, Ref. 25.

[30] Bailin, *op. cit.*, Ref. 22; Blaha, et al., *op. cit.*, Ref. 22; Henderson-Sellers and Edwards, *op. cit.*, Ref. 22; Hull and King, *op cit.*, Ref. 29; McFadden and Hoffer, *op. cit.*, Ref. 22; Papazoglou and Marinos, *op. cit.*, Ref. 29; Peckham and Maryanski, *op. cit.*, Ref. 22.

[31] Goldberg and Robson, *op. cit.*, Ref. 20; Ricardo, C. (1990). *Database Systems: Principles, Design, & Implementation.* New York, NY: Macmillan Publishing Company, inc.

[32] Peckham and Maryanski, *op. cit.*, Ref. 22.

[33] Blaha, et al., *op. cit.*, Ref. 22; Henderson-Sellers and Edwards, *op. cit.*, Ref. 22; Hudson and King, *op. cit.*, Ref. 28; Papazoglou and Marinos, *op. cit.*, Ref. 29; Unland and Schlageter, *op. cit.*, Ref. 25.

[34] Goldberg and Robson, *op. cit.*, Ref. 20; Ricardo, *op. cit.*, Ref. 31; Unland and Schlageter, *op. cit.*, Ref. 25.

[35] Schoderbek, Schoderbek, and Kefalas, *op. cit.*, Ref. 4, p. 18.

[36] Cox, *op. cit.*, Ref. 28; Fishman, et al., *op. cit.*, Ref. 28; Gibbs, et al., *op cit.*, Ref. 25; Hammer and McLeod, *op. cit.*, Ref. 22; Howard, *op cit.*, Ref. 25; Jones, J. H. (1989) . Object Properties of ANS '85 COBOL. In *Proceedings of Seventeenth Annual North American Conference International Business Schools Computer Users Group*, July 1989; McFadden, F. R. and McIntyre, S. C. (1990). Intelligent Databases. In *Proceedings of Decision Sciences,* November

1990, pp. 1122-1124; McIntyre, S. C. and Higgins, L. F. (1988). Object-Oriented Systems Analysis and Design: Methodology and Application. *Journal of Management Information Systems, 5* (No. 1), pp. 25-35; Peckham and Maryanski, *op. cit.,* Ref. 22; Unland and Schlageter, *op. cit.,* Ref. 25; Wirfs-Brock, R. J. and Johnson, R. E. (1990). Surveying Current Research in Object-Oriented Design. *Communications of the ACM, 33* (No. 9), pp. 104-124.

[37] Agha, *op. cit.,* Ref. 25; Cox, *op. cit.,* Ref. 28; Gibbs, et al., *op. cit.,* Ref. 25; Korson and McGregor, *op. cit.,* Ref. 25.

[38] Harrison, W. H., Schilling, J. J. and Sweeney, P. F. (1989). Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. In *Proceedings OOPSLA '89 SIGPLAN Notices, 24* (No. 10), pp. 85-94; Sciore, *op. cit.,* Ref. 25; Unland and Schlageter, *op. cit.,* Ref. 25.

[39] Gibbs, et al., *op. cit.,* Ref. 25; Goldbert and Robson, *op. cit.,* Ref. 20; Hudson and King, *op. cit.,* Ref. 28; Sciore, *op. cit.,* Ref. 25.

[40] Banerjee, et al., *op. cit.,* Ref. 28; Goldberg and Robson, *op. cit.,* Ref. 20; Lerner and Habermann, *op. cit.,* Ref. 25; Wirfs-Brock, A. and Wilkerson, B. (1989). Variables Limit Reusability. *Journal of Object-Oriented Programming,* May/June 1989, pp. 34-40.

[41] Blaha, et al., *op. cit.,* Ref. 22; Hammer and McLeod, *op. cit.,* Ref. 22; Hull and King, *op. cit.,* Ref. 29; McFadden and McIntyre, *op. cit.,* Ref. 36.

[42] McFadden and Hoffer, *op. cit.,* Ref. 22; Unland and Schlageter, *op. cit.,* Ref. 25.

[43] Schoderbek, Schoderbek, and Kefalas, *op. cit.,* Ref. 4, p. 19.

[44] *Ibid.,* p. 23.

[45] Korson and McGregor, *op. cit.,* Ref. 25; Maier, *op. cit.,* Ref. 22; Wirfs-Brock and Johnson, *op. cit.,* Ref. 36.

[46] Loomis, M. E. (1990). The Basics. *Journal of Object-Oriented Programming,* May/June 1990, pp. 77-81; Maier, *op. cit.,* Ref. 22.

[47] Loomis, *op. cit.,* Ref. 46; Wirfs-Brock and Wilkerson, *op. cit.,* Ref. 28.

[48] Maier, *op. cit.,* Ref. 22.

[49] Hammer and McLeod, *op. cit.,* Ref. 22; Hull and King, *op. cit.,* Ref. 29; Peckham and Maryanski, *op. cit.,* Ref. 22; Wirfs-Brock and Johnson, *op. cit.,* Ref. 36.

[50] Agha, *op. cit.,* Ref. 25; Howard, *op. cit.,* Ref. 25; Jones, *op. cit.,* Ref. 36.

[51] Cox, *op. cit.,* Ref. 28; Howard, *op. cit.,* Ref. 25; Korson and McGregor, *op. cit.,* Ref. 25; Wirfs-Brock and Wilkerson, *op. cit.,* Ref. 28.

[52] Cox, *op. cit.,* Ref. 28.

[53] Cox, *op. cit.,* Ref. 28; Gibbs, et al., *op. cit.,* Ref. 25; Wirfs-Brock and Wilkerson, *op. cit.,* Ref. 40.

[54] Cox, *op. cit.,* Ref. 28.

[55] *Ibid.*