

Structural (Cognitive Task) Analysis: An Integrated Approach to Software Design and Programming

Joseph M. Scandura, Ph. D.
MERGE Research Institute, Scandura.com
and University of Pennsylvania

Abstract

The goal of this paper is to introduce a method of (domain) analysis that has long been used in analyzing cognitive domains. Over the past few years, structural (cognitive task) analysis has been refined, extended and successfully applied in software design and programming. Structural analysis (referred to as Cognitive Object Oriented (COO) design to avoid confusion with structured analysis) makes it possible: (a) to model “real world” input-output behavior more directly than in the standard OO paradigm, (b) to test and debug OO models as they are being designed, (c) to directly support (as opposed to just enabling) polymorphic operations as well as class hierarchies, (d) to be implemented in C++ (and other OO languages such as Java) and (e) to support code reuse, including the reuse of legacy code. Reference also is made to AutoBuilder, an integrated software engineering system that supports many of the ideas introduced.

Software design and programming require overlapping but different skills. Both place heavy emphasis on assembling components to achieve desired ends. The ability to abstract, however, is more essential in software design whereas programming requires more attention to numerous details. These differences are often reflected in practice, and characterize the methods used. While programs necessarily address all requirements for execution, this has not been the case for design. Structured analysis, information modeling and more recently the object-oriented (OO) paradigm all emphasize different aspects of the process. Structured analysis and state transition diagrams, for example, emphasize top-down analysis of processes, but they implicitly assume all inputs and outputs are known from the beginning. Conversely, information modeling deals similarly with data but says little about processes.

OO modeling emphasizes the importance of partitioning “real-world” models into objects. OO accommodates both data and processes, and has the additional advantage of strongly encouraging code reuse. Indeed, most software experts no longer question whether OO will replace traditional approaches to software development. It is more a matter of how quickly one should move to OO, and to a
lesser extent how “pure” one should be. Although OO design methods are still evolving (e.g., Bruno, 1995), many of the better known OO methodology authors (e.g., Booch, 1994; Rumbaugh et al., 1991; Coad & Yourdon, 1991) are in general agreement as to how software systems should be designed and implemented (e.g., UML, also see Orfali, 1994; Palmer, 1994; Rogers, 1994; Wood, 1994).

In spite of its advantages (e.g., improved modularity and reusability), however, the OO paradigm has a number of important weaknesses. One important problem involves how to combine objects in producing large systems. Mutual dependencies complicate flow of control making it difficult to debug complex OO applications. System behavior is easily obscured in message passing and unconstrained inheritance. Although the subject of on-going research, this problem is not considered herein.

A problem that is addressed herein has to do with the way polymorphic and overloaded operations are handled. Because types in class hierarchies differ, operations performing similar functions are normally referenced differently in writing code. To perform a common operation on differing objects associated with a class hierarchy, one typically references both the (e.g., polymorphic) operation and a particular (sub)class. Although this is not a problem for those skilled in OO methods and programming, it does impose an extra cognitive load on the designer and/or programmer. In talking about cleaning rooms, for example, it is more natural to say or write “clean (room)”, irrespective of the type of room.

In effect, viewing “objects” as both star and supporting cast has the effect of reducing the importance of operations. Although organizing functions by data type has certain advantages, it also has disadvantages. Operations having little conceptually to do with one another are, nonetheless, in practice often grouped together.

In the real world, actions play as important a role as objects. At a minimum, one needs both “nouns” and “verbs.” One cannot express a complete thought (i.e., write a sentence) without using or implying both an object (noun) and an action (verb). In OO, actions are handled by functions within objects. That is, they are physically grouped by data type rather than by semantic equivalence. Verbs, on the other hand, correspond more closely to polymorphic operations. They can be viewed as single abstract operations acting on a variety of data types. Making functions subordinate by putting them inside objects (where they act in an automated fashion) makes them conceptually more like direct perception rather than human cognition - more like modifiers (adjectives?) than verbs.

As is well known, the number of tests required for verification goes up exponentially with system complexity (Scandura, 1992). Consequently, the traditional design-implement-test paradigm -- whether OO or otherwise -- is inadequate as regards verification. If it were possible to truly test and debug designs at higher levels of abstraction, one could reduce testing to manageable proportions. (The ability to ensure logical

1The author wishes to thank Dr. Mort Hyman, Thomas Honigmann and several unnamed reviewers for their helpful comments on a draft of this article. The author is fully responsible for remaining inadequacies.

2Except where used in connection with C (or Ada), the term “function” is used here in a generic sense to refer to any operation.
consistency of models as they are being developed, of course, would be even better.)

Purpose.-- The purpose of this paper is to introduce the method of structural (cognitive task) analysis\(^3\) to the software engineering community. (The “at” is intended because structural analysis was developed independently of Wirth’s (1976) contemporaneous structured programming and structured analysis.) Structural analysis has a long history in the cognitive sciences (e.g., Scandura, 1971, 1973, 1977, 1982, 1984), and derives from work reported at a decade-long series of international, interdisciplinary structural learning conferences held between 1968 and 1977.

Structural analysis was originally designed to analyze cognitive domains from a cognitive point of view. It is based directly on real-world semantics, and includes both abstract operations and abstract objects. In recent years the method has been the subject of on-going research (with parallel development). This research is designed to refine, extend and apply structural analysis to software design and programming.

This paper represents a work in progress. In it, I show how structural analysis makes it possible: (a) to model “real world” input-output behavior more directly than in the standard OO paradigm, (b) to test and debug OO models as they are being designed, (c) to directly support (as opposed to just enabling) polymorphic operations as well as class hierarchies, (d) to be implemented in C++ (and other OO languages such as Java) and (e) to support code reuse, including the reuse of legacy code. To avoid confusion with structured analysis, structural (cognitive task) analysis is referred to in this paper as Cognitive Object Oriented (COO) design.

Reference in the paper also is made to AutoBuilder, an integrated software engineering system that supports many of the ideas introduced.\(^4\)

Cognitive OO Design

All systems can be modeled in terms of input, output and operations which map the first onto the second. The essence of the cognitive OO paradigm (alias COO/structural analysis) involves modeling system behavior successively from the highest levels of abstraction. Real world terminology is introduced as needed to represent desired operation and data constructs; the process continues until contact is made with available data and computational resources. Artificial barriers between high and low level design and programming are eliminated.

At the top level of analysis, all system modeling begins by labeling the abstract input, output and operation. To emphasize the point, consider a complex software engineering application (such as AutoBuilder). At the highest level of abstraction, for example, input might simply be the "user_intent". (At a more detailed level, of course, this intent is reflected in more explicit input.) Similarly, the initial output refers to the collective outputs of the application — say “software_engineering_outputs.” The top

---

\(^3\) The terms structural (cognitive task) analysis, cognitive analysis and cognitive OO (COO) are used largely interchangeably throughout.

\(^4\) A novel method based on COO ideas, which automates key design processes while insuring correct software, is disclosed in a recent patent application.
level operation, then, might be called “AutoBuilder”.

Clearly, at this level of abstraction, little is known about the application other than its domain of applicability, and then only in a very generic sense. Notice, however, that abstract inputs (i.e., “user_intent”) and outputs correspond to virtual objects, much like a base class in C++ or Java. Indeed, we will find operations lurking inside the objects as we probe more deeply. The abstract operation “AutoBuilder,” however, is different. It acts on and produces objects as wholes, with only implicit reference to specific procedures or functions contained in objects (or equivalently to passing “messages” to the objects).

In modeling behavior, structural (cognitive) analysis shares much in common with structured analysis. Paralleling structured analysis, for example, operations can be refined into a sequence, a selection (e.g., IF...THEN) or iteration (e.g., LOOP). One can also envisage parallel execution. In structured analysis, however, data is not refined (as it is say in information or data analysis). It is simply assumed to be available as needed. In contrast, data refinement is an explicit part of cognitive analysis. Abstract data must be refined in parallel with abstract operations. We return to this matter after discussing process refinement.

Process Refinement.-- If a parent process is refined into a sequence, then the child processes use the same abstract inputs and outputs as the parent. If the refinement is a selection, however, we must introduce an “extra domain” condition variable or relation that partitions the domain. Domain elements in one equivalence class of the partition serve as input to the THEN statement. Inputs that fall in the other equivalence class of the partition are input to the ELSE statement. Clearly, selections can be generalized to CASE structures involving any number of elements in the partition. Since the output of the original abstract operation must reside in the abstract range variable it follows that the output of each CASE alternative must also lie in that same range variable.

The case of iteration can be viewed as a variation on binary selection in which the THEN statement, or LOOP body, accepts inputs in both partition elements. In turn, outputs of the LOOP body must be legal values of the abstract domain variable with elements in both equivalence classes of the partition. The hard part is to ensure that each iteration of the LOOP body modifies outputs in a systematic way so as to ensure a transition between the equivalence classes in the domain partition (one of which exits the loop).

One might also refine an operation into two or more operations, which act in parallel on the domain and collectively produce elements in the range.

Data Refinement.-- Except for terminology and perhaps emphasis, cognitive analysis so far appears identical to structured analysis. Structured analysis presumes concrete inputs and outputs. In contrast, every cognitive analysis begins with a virtual input, output and process.

Abstract inputs clearly do not provide the detail necessary to reflect the data complexity required for implementation. Data structures are introduced gradually in parallel with process decomposition. The equivalence classes introduced by a selection partition, for example, correspond to new domain variables, in particular to subsets of the original abstract domain. Abstract inputs and outputs become
increasingly concrete as a result of refinement, providing more and more information about the data structures involved. Eventually, refinement of virtual data makes contact with real data.

What is essential in cognitive analysis is that data and process remain in sync at each level of refinement. This is true whether talking about virtual data and processes or real ones. (We see below that this coordination of data and process has important implications for testing and debugging design models.)

Data refinement leads naturally to hierarchical data structures, analogous to record structures. (Arrays in this context are simply special cases.) Other kinds of relationships also arise naturally as a result of the "extra domain" conditions introduced to partition abstract data elements. In particular, conditions may be defined as functions on other data. Moreover, concrete data structures may be derived from abstractions -- much as in inheritance. Conversely, abstract data may be derived in terms of the more concrete -- often called "evaluation." Defining conditions in terms of other data comes up naturally in the following example.

**Example.**-- Irrespective of the system being modeled, cognitive analysis always begins with INPUT, OUTPUT and OPERATION (see Figure 1A). For illustrative purposes, consider

```
[FIGURE_1.FLX] :name_of_model;[]library;[procedure];

[INPUT]:

[OUTPUT]:

OPERATION
```

*Figure 1A. The common starting point for modeling any system. "Figure_1.FLX" is the name of the FLexform file, "name_of_model" is the model name, "library" is the name of our extensible high level design language and "procedure" indicates the type of FLexform. Other symbols are strictly syntactic and may be ignored.*
the simple task of cleaning a room. (The example might be suggestive if you have young children at home, or are into robotics.)

The first step is to name the top level INPUT, OUTPUT and OPERATION abstractions (see Figure 1B). Notice that the top level abstraction under OPERATION is a full statement referencing both “room” (which in this example serves as both input and output) and the operation “clean.” The terminology used in this simple High Level Design (HLD) language is completely arbitrary and can be chosen as desired to reflect the underlying semantics.

**Simulation.** -- AutoBuilder software includes an HLD Simulator which supports dynamic “execution” of HLD designs. High level designs can be executed interpretively at any level of abstraction. Test coverage and various debugging modes also are supported. Execution is automatic where referenced components have been semantically wrapped and integrated into the AutoBuilder software. Virtual (abstract) components are simulated manually. The user is shown the current status of the data and is asked to perform the virtual operation and update the data. Both real and virtual components interoperate so designs can be tested (and debugged) at successive levels of detail through implementation. In the process, elements in one design module can call other modules, with explicit support for recursion.

In particular, the high-level clean room specification can be run as represented in the HLD Simulator. At the beginning of execution, the value of “room” is “unpresentable” as shown. Execution of this high level model involves only the operation “clean.” During execution, the cursor stops at the operation, the current value of “room” is displayed (i.e., “unpresentable”) and the user is asked to perform (simulate) the “clean” operation and update the value of room. In this case we might enter “presentable.”

5 See www.scandura.com for more details.
**Sequence Refinement.**-- In Figure 2, “room” is refined into, and defined in terms of, the components, “bed” and “carpet”. As detailed below, the value of room is a function of bed and carpet. Correspondingly, the behavioral design is refined into a (possibly parallel) sequence of two operations: make the bed and vacuum the carpet. Initially, the bed is unmade and the carpet is dirty. After execution, the bed might be made and the carpet, clean. (The latter is not shown in Figure 2.)

**Loop Refinement.**-- Figure 3 shows what happens when we refine “vacuum (carpet)” into a loop. In this case, we “advance_and_vacuum” the carpet in increments: First “seg1,” then “seg2” and finally “seg3.” In effect, refinement of the operation “vacuum” (in the statement “vacuum (carpet)”) into a loop requires that carpet be partitioned into segments. Notice that “carpet” has been defined as the “union” of three segments -- a classic case of evaluation. The value of carpet is a function defined on values of the three rug segments. If all of them are clean then the carpet if clean. Otherwise, carpet is dirty. Also notice that the loop requires introduction of an alias (a pointer), providing a common reference to the various segments (one at a time).

**Loop Simulation.**-- In this case, execution begins as before with “make (bed).” Then, the user is told to

```plaintext
[COO_2.flx] :clean_room;[library;[procedure];
[ INPUT ]:
[ room]:unpresentable
[ bed]:unmade
[ carpet]:dirty

[OUTPUT]:
[ room]:presentable

x make (bed)
x vacuum (carpet)
```

*Figure 2. Sequence refinement of "clean (room)" into "make (bed)" followed by "vacuum (carpet)." The procedural refinement indicates that "make" and "vacuum" may be parallel operations.*
involves first determining what is to be aliased, and then updating the value of the referenced element (e.g., seg1).

The loop condition is simply a matter of evaluating "done". "done" is an example of what we called earlier an "extra domain" condition. It partitions the carpet segments into two equivalence classes, one that includes segments (seg1, seg2) and returns control to the loop body and one (seg3) which exits the loop. The value of "done" might be determined by a function (called "next_element_exists") defined on the alias. Specifically, "next_element_exists (current_segment)" automatically determines whether we are "done" (i.e., whether the next element is nill). We call this "data execution". One must execute the loop body three times before the condition is true.

Incidentally, "next_element_exists" is just one of the semantic core functions built into the Simulator which provide a full set of semantic model building facilities. Execution of "done" in this case is automatic.

Further Refinement.-- Figure 4 shows further refinement of the clean room model. In this case the loop body has been refined first into a sequence, then the second operation into a selection and finally the THEN alternative into a sequence. The first statement in the body involves "next_element," another of the core semantic functions. Each time through the loop, this statement automatically assigns the next segment to current_segment. The condition "match" checks to see if current_segment is an alias for a segment that is "messy & dirty." (Seg2 is the only segment with this value.) In effect, the selection refinement partitions the segments into two equivalence classes, those that are "messy & dirty" and those that are not. When the THEN sequence is executed, the "pickup" operation changes the value of current_segment to just "dirty" -- that is, it gets rid of the "messy" part. The operation "vacuum" does the rest (i.e., it cleans seg2).

Data-Process Abstraction and Design Debugging.-- Notice the natural progression inherent in the process. Abstract operations and objects can be refined successively and indefinitely. There is no natural or required transition between high level and detailed design. Indeed, as noted below, there is no sharp distinction between detailed design and programming, other perhaps than shifting from a generic representation to a specific one (e.g., to a programming language like C++ or Ada 95). Successive phases of refinement flow naturally from one to the other. Any divisions introduced between high level analysis and detailed design are arbitrary and strictly for convenience.
The fact that data (objects) and process (operations) must be represented at the same compatible level of abstraction has important implications for testing -- in two words “design debugging.” Abstract objects are operated on by correspondingly abstract operations. Consequently, the HLD Simulator (above) can execute any such model at arbitrary levels of abstraction. This makes it possible to simulate and debug systems as they are being developed. (We shall see below how data refinement relates to OO inheritance.)

Scandura has shown elsewhere (1990, 1992) that debugging designs from the highest levels of abstraction dramatically reduces the number of empirical tests required to “prove” a system. When testing is delayed until after a system has been implemented, the number of paths to be tested goes up exponentially with complexity. This number only goes up additively when testing is done successively from the highest levels of abstraction.

To summarize, the major difference between the above and traditional structured analysis and information modeling is that structural (cognitive task) analysis requires parallel refinement of both processes and data. Unlike structured analysis, process refinement (sequence, selection, iteration) imposes parallel requirements on virtual data. In addition to simple decomposition, data objects (elements) and their values can be defined in terms of functions on other elements. Notice that the same (or equivalent) behavior is determined

---

**Figure 3. Illustrates refinement into a loop.**

```plaintext
[COO_3.flx]: clean_room;[]library;[procedure];
  [INPUT]:
    @ [room]:unpresentable
    [bed]:unmade
    [carpet]:unclean
    [seg1]:dirty
    [seg2]:messy&dirty
    [seg3]:dirty

[OUTPUT]:
  @ [room]:unpresentable

[Intermediate]:
  [current_segment](alias):

  [done]same (current_segment, seg3):

  x make (bed)
  x REPEAT advance_and_vacuum (current_segment)
  x UNTIL done
```
irrespective of the level of data and/or process abstraction used in a simulation or execution. For example, “clean (room)” results in the same behavior -- a presentable room -- as does “make (bed)” and “vacuum (carpet)”. A room is defined to be unpresentable when either the bed is unmade or the carpet is not clean, and presentable when the bed is made and the carpet is clean. (Notice that "make" has nothing to do if the bed is already made, and similarly with "vacuum".) Given this functional relationship, the parent operation, clean, performs the same behavior as do the subordinate operations make and vacuum. In effect, maintaining compatibility between data and process at successive levels of refinement makes it possible to test and debug software as it is being designed -- in a manner which is far more efficient than the traditional design-implement-test paradigm (patent pending).

As we shall see below, data objects (as well as their values - cf. "done" above) can be constructed dynamically by functions defined on other objects.

**Extension to Object Inheritance.** So far, we have assumed that the virtual components used in designs, as well as components used to implement designs, are defined on static data structures. No behavior was ascribed to the data structures themselves. The terminal objects (e.g., “bed”, “carpet”) comprising these structures are typeless. (As noted above, data structures can be defined in terms of other data via data execution. However, such behavior is equivalent to using embedded functions; it is between objects, not within objects.)

Inheritance in the OO paradigm is an "is a" relationship between categories of objects. A bedroom, for example, is a category of room. Compare this with the above clean room example where "room" was defined as consisting of two parts, a bed and a carpet (i.e., as an "is part of" relationship). The internal structure of room may or may not be made explicit (e.g., it might be defined in terms of wall, floor and ceiling components). To be a bedroom in either case requires additional structure (e.g., beds and carpets). Categories of rooms, say bedrooms and kitchens, necessarily include both the components of rooms in general and the components which distinguish those categories. Thus, bedrooms typically contain beds and carpets. Similarly, kitchens contain stoves and sinks.

In traditional OO, rooms and bedrooms correspond to separate, albeit hierarchically related classes (types with associated functions defined on those types). In the real world, however, objects do not correspond directly to such classes. Rather, bedrooms for example include the components associated with both room and bedroom classes. Because of this disjunction in meaning, OO terminology can easily lead to confusion and become a source of conceptual errors in design.
When we speak of cleaning rooms from a real-world semantic (cognitive) perspective, reference is made to all rooms. Making beds and vacuuming carpets are integral parts of the process when the room in question happens to be a bedroom. But, this level of procedural detail (in a design) is not necessary unless we are talking explicitly about beds and carpets. When we say “clean (room)”, and the input room is a bedroom, the intent is to make the bed and vacuum the carpet, as well as to clean the generic parts of the room. Real-world cognitive semantics calls for an abstract, albeit executable operation which operates on all rooms. Such a clean operation implicitly includes make and vacuum.

This raises the question of how best to implement a procedural statement like “clean (room)” when functions like make and vacuum (in subordinate objects) are affiliated with clean (a function in a superordinate object). In most OO languages, such affiliation is most commonly handled by introducing overloaded clean operations. In this case, when a room object is passed to clean, the program must decide which clean to invoke. The choice is typically

```
[COO_4.flx] :clean_room;[]library;[procedure];

[ INPUT ]:
  @[ room]:unpresentable
  [ bed]:unmade
  [ carpet]:dirty
  [ seg1]:dirty
  [ seg2]:messy&dirty
  [ seg3]:dirty

[OUTPUT]:
  @[room]: unpresentable

[Intermediate]:
  [ current_segment](alias):

  [ done]same (current_segment, seg3):
    x make (bed)
    x REPEAT current_segment = next_element (carpet, current_segment)
    x IF match ( current_segment, 'messy&dirty')
    x THEN  pick_up (current_segment)
    x vacuum ( current_segment)
    x ELSE vacuum ( current_segment)
    x UNTIL done

Figure 4. Further refinement of the clean room model.
determined by the type of the room object. In typeless OO languages, the same result can be obtained by choosing the function whose domain most fully matches the object’s structure. The clean associated with the bedroom class would be chosen when the room in “clean (room)” is a bedroom -- because it offers a better fit than does the original clean (i.e., it deals with beds and carpets as well as generic room components).

A generic clean operation might be implemented, for example, by introducing case alternatives (within the operation) which anticipates everything that rooms might contain. Alternatively, the case alternatives might refer to the kinds of clean sub-operations which need to be performed (e.g., on objects having hard impervious services, on cloth-like items to be arranged symmetrically, etc.).

The same result can be accomplished in a way that more naturally reflects real-world semantics -- by introducing the notion of affiliate functions. In modeling system behavior at a high level of abstraction, clean is assumed to operate on all room objects. From a real-world behavioral perspective, it is an external operation (and not just one of potentially many functions associated with room objects). “Clean (room)” means: Clean all of the components which define room (e.g., walls and floor) plus all of its contents (e.g., beds, carpets or whatever).

This affiliation concept can be implemented directly by associating functions in subordinate classes with operations in (or operating on) superordinate ones. When we add a function, like make or vacuum, to a subordinate class, we would be given the option of affiliating that function with functions associated with the superordinate class. In this case, calling “clean (room)” would have the effect of (also) executing those of its affiliates that are associated with any given room object.

The latter implementation, of course, might retain the option of specifying functions associated with particular classes -- perhaps using the common syntax: object.function (data). For example, “bedroom.clean (room)” might be used to specify a particular instance of an overloaded clean function. (NOTE: From a behavioral perspective, there is little difference between calling a function in a specified object and calling one in a function library.) The flexibility inherent in the latter syntax might be deemed essential by many OO programmers. Nonetheless, I suspect anything that can be done using the extended syntax can also be done in the proposed semantically more palatable manner via better design -- much like GOTOs can be eliminated in structured programming.

Discussion of the OO Paradigm

In an earlier article (Scandura, 1997) showed how COO can be viewed as part of a natural evolution of programming paradigms, from procedural through OO to COO Programming.

In this context a distinction was made between enabling a paradigm and supporting one. A language supports a programming style if language facilities make it convenient, reasonably easy, safe and efficient to use that style. If it takes exceptional effort or skill to write programs in a given style, then that language merely “enables” the technique (e.g., see Stroustrup, 1991). For example, one can write structured
programs in FORTRAN and object oriented programs in C, but it is difficult to do so because these languages do not directly support those paradigms.

The object oriented paradigm allows new classes to “inherit” from existing, superordinate classes. Functions in those superordinate classes are automatically defined on objects in the subordinate classes. In addition, conceptually similar functions in different classes may be overloaded (i.e., use the same name but operate on different data structures). Polymorphism ensures that the appropriate function is invoked dynamically at run time based on object types. Procedure calls between objects are accomplished by “message passing” (the OO equivalent to procedure calling). Data structures in classes are often but not necessarily defined statically. They also may be derived from generic class/package definitions -- as in Ada generic packages and C++ template classes.

Limitations of OO Programming.-- In OO programming, inheritance refers to classes one way and to objects in another. Subtypes in subordinate classes supplement superordinate ones. Objects, on the other hand, include data structures associated with both superordinate and subordinate classes. For example, whereas a bedroom class refers only to structures unique to bedrooms, (bedroom) objects associated with that class include structures associated with both bedroom and room classes. This can easily lead to confusion or unnecessary complication in modeling real-world semantics.

Similarly, method (function) overloading (and/or polymorphism) typically involves distinct functions having the same names in different classes. Real-world semantics, on the other hand, typically calls for single all-encompassing operations which combine the effects of all (overloaded/polymorphic) functions sharing a name. Behaviorally speaking, such all-encompassing operations operate on objects as wholes. Putting them inside objects does not conform to real-world usage (all the more so for those not familiar with OO constructs).

By distorting this simple semantics, standard OO implementations can easily lead to unnecessarily complicated and hard to maintain code. Rather than simply writing one clean operation (with affiliates), which refers to all rooms (bedrooms and otherwise), for example, one would have to maintain several clean operations, one for each category of room.

An important difference between objects as resources and the use of objects in modeling behavior also is often neglected. Collecting functions inside objects because they use a common data type is reasonable from the perspective of defining resources. In modeling behavior, however, operations become paramount. In this case, it is cognitively more palatable to view different (albeit hierarchically related) data types according to the operations that use them. (Indeed, the increasingly recognized importance of mini-uses and use-cases -- as exemplified in on-going attempts to clarify these informal notions in UML-based OTUG discussions -- also suggest that abstract operations may be given insufficient attention in OO modeling.)

In short, semantic differences (between classes used in inheritance hierarchies and objects associated with those hierarchies and between
overloaded/polymorphic operations and cognitive intent) can easily lead to unnecessary complications in modeling behavior. Reliability, availability (understandability) and maintainability (RAM) are reduced as a result. Designs and programs become unnecessarily complex; debugging and testing can become even harder. To overcome these limitations, abstract all-encompassing operations, which incorporate affiliate operations, are introduced. Functions within objects may be referenced by (i.e., become affiliates of) these abstract operations but they are conceptually different. The former abstract operations act on sets of objects, irrespective of type; the latter operate on components from which these objects are constructed.

Cognitive Object Oriented (COO) Programming Paradigm.-- The COO paradigm supports abstract (albeit executable) operations which operate on sets of input objects and which generate output objects. COO operations are based directly on real-world semantics. They accomplish their work by referencing affiliated functions associated with classes in inheritance hierarchies (used in defining object parameters).

In addition to functions within objects and abstract operations defined on objects, COO supports associating functions directly with objects. The latter serve to define given objects or their values (dynamically) in terms of other objects, much like functions embedded in procedure calls. We call this “data execution”. The only difference between data execution and any other type of function is that these functions are associated directly with data, rather than inserted in procedural logic. Data execution simplifies procedural logic, and localizes the definition of new objects. In the original clean example (Fig. 2), for instance, room is defined directly in terms of bed and carpet. That is, the value of room is determined by the values of bed and carpet. Similarly, in designing an algorithm for simulating the way a child might perform column subtraction, data execution might be used to define, say, top-digit as the common element of top-row and current-column. Instead of having to define top-digit repeatedly in writing the algorithm, data execution makes it possible to do it just once.

COO Analogy to Cognition.-- The distinction in COO programming between abstract operations (which act on objects as a whole) and functions used to define objects has direct parallels in human cognitive processing. Functions associated with objects correspond rather directly to automated (well-learned, perceptual) processes, actions that take place more or less automatically without conscious thought. In the clean room example, one knows when one is “done.” The functional definition is executed automatically as required.

More generally, functions effectively define relationships between objects in a hierarchy. Data execution allows objects (e.g., rooms) to be defined dynamically in terms of other objects (e.g., beds and carpets). Data execution in COO is a modifier, serving a purpose analogous to adjectives (which modify nouns).

Operations between input and output objects refer to more conscious cognitive processing. In the real world we act on objects: We write, speak, shave, drive and eat. We do not just pass “messages” to objects telling them how to behave. Writing programs that reflect the way we
naturally describe behavior has obvious advantages as regards both construction and understandability.

In effect, the COO paradigm provides a more complete characterization of the "real world", including abstract operations (characterizing the abstract essence of polymorphic operations) as well as objects. This is analogous to a sentence having both a noun and a verb. (Functions defining objects are analogous to adjectives modifying nouns.) Cognitive programs are a natural consequence of COO analysis and design, and may be tested and debugged at arbitrary levels of refinement from the highest levels of design through implementation. A key concept is that the collective behavior of child operations in every refinement must be equivalent to that of the parent. Demonstrating equivalence in this case requires reference to functions defining relationships between parent and child objects (e.g., between room, and bed and carpet), as well as between inputs and outputs.

**COO Design-Programming Continuum.**--
As shown in the clean room example, COO programming is a natural extension of COO design. Like COO design, COO programming directly reflects the way people describe the real world. The contents of Flexform elements, as well as overall Flexform relationships (both between Flexform elements and between different Flexforms -- cf. Scandura, 1987, 1990, 1992, 1994), are of the same form whether one is talking about design or programming.

There are, of course, important differences. Terminal elements in Flexforms at the implementation level correspond to available resources -- to data structures, functions and/or operations that are or can automatically be converted into machine-readable form. These resources may correspond to native statements in a programming language, operating system services, or objects, functions or procedures in a library. Whereas the designer has complete freedom during design to specify exactly what (virtual components) are to be used, the programmer is constrained to use available executable components. The executables must be assembled so they meet specifications defined by the corresponding (terminal virtual) components in associated designs.

**More Complete Representation of the "Real World."**--
The above discussion led naturally to abstract objects (having sets of types), abstract operations (and their affiliates on components of abstract objects) and (automatic) functions between objects. Nouns, verbs and adjectives were shown to be real world analogues. This raises the question of adverbs. Does anything in COO programming correspond to verb modifiers (i.e., adverbs)?

The answer is a qualified yes. As originally conceived, structural analysis (e.g. Scandura, 1971, 1973, 1982, 1984) is a recursive process. The first step identifies sets of individual solution rules (alias programs) associated with the domain.

In human problem solving, the domain may be so large, however, that it is difficult to devise any finite set of programs which provide solutions for all problems in the domain. Structural analysis was designed to handle this situation (e.g., Scandura, 1971, 1973, 1984). Given an initial (lower order) set
of solution rules, structural analysis builds on this base and offers a recursive method for identifying higher order rules (analogous to higher order operations/programs).

These higher order rules take (other) rules as input and generate rules as outputs. The input/output rules are of lower order (relative to the higher order rules). The important point is that higher and lower order rules operating collectively can generate solutions for (often much) broader ranges of problems than those same rules used individually. The power of the approach derives from often dramatic increases in problem solving potential at successive levels of structural analysis (e.g., Scandura, Durnin & Wulfeck, 1974; Scandura, 1977). Higher order operations in this context effectively modify other operations -- analogous to “adverbs” modifying “verbs.”

In effect, just as data execution supports dynamic construction and/or modification of objects (i.e., data types), higher order operations support dynamic construction and/or modification of operations. This latter idea has been frowned on in traditional programming because undisciplined, dynamic modification of operations often results in hard to detect side effects. When used in a highly disciplined manner, however, cognitive research strongly suggests that this type of “higher order” programming could add a degree of controlled power rivaling the typically more idiosyncratic use of AI techniques. In this context, higher order components appear to share some things in common with self-modifying intelligent software agents.

There are no global variables as such in COO design. Hierarchical data structures dramatically reduce the number of parameters required. These parameters may take objects having different types as values. Objects themselves may contain functions, which serve to define the objects dynamically in terms of other objects.

It is of some interest to see how COO designs may be implemented in an OO language like C++ or Java. Accomplishing this requires two things: (a) a way to define values and/or types of objects in terms of values and/or types of other objects and (b) a way to accommodate abstract operations which operate on values of objects whose types may differ.

(a) COO programming allows “real world” objects to get their values from, even to dynamically be constructed from other objects. In the above example, the room was defined as a function of bed and carpet.

COO also supports the dynamic definition of object data structures (types) in terms of the types of other objects. That is, data structures can be dynamically constructed from other structures. For example, bedroom can alternatively be defined as a function of (generic) room, bed and carpet. In COO, the term data execution encompasses both deriving values of objects (variables) from functions of other objects and deriving types of objects from the types of other objects. Data execution is more general than inheritance in this sense. In the latter case, subordinate structures are defined solely by aggregation (e.g., adding beds and carpets to generic rooms).

Implementation of COO Programming in Java

There are no global variables as such in COO design. Hierarchical data structures dramatically reduce the number of parameters required. These parameters may take objects having different types as values. Objects themselves may contain functions, which serve to define the objects dynamically in terms of other objects.

It is of some interest to see how COO designs may be implemented in an OO language like C++ or Java. Accomplishing this requires two things: (a) a way to define values and/or types of objects in terms of values and/or types of other objects and (b) a way to accommodate abstract operations which operate on values of objects whose types may differ.

(a) COO programming allows “real world” objects to get their values from, even to dynamically be constructed from other objects. In the above example, the room was defined as a function of bed and carpet.

COO also supports the dynamic definition of object data structures (types) in terms of the types of other objects. That is, data structures can be dynamically constructed from other structures. For example, bedroom can alternatively be defined as a function of (generic) room, bed and carpet. In COO, the term data execution encompasses both deriving values of objects (variables) from functions of other objects and deriving types of objects from the types of other objects. Data execution is more general than inheritance in this sense. In the latter case, subordinate structures are defined solely by aggregation (e.g., adding beds and carpets to generic rooms).
Abstract operations play a central role on COO design and programming. They operate on objects as wholes -- irrespective of the complexity of the class hierarchy corresponding to these objects -- combining the effects of all semantically related functions (not necessarily with just those having the same name). Thus, our clean room example might be extended by defining the abstract operation "clean" in terms of affiliates like "make", "vacuum", "wash", etc. Where the room in "clean (room)" is a bedroom, for example, clean would have the effect of cleaning generic parts of the room plus making the bed and vacuuming the carpet. Where the room is a kitchen, it would clean the generic parts plus the stove and sink.

As we saw above, abstract operations combine the effects of affiliated functions in an inheritance hierarchy. Overloaded functions are just one example of affiliate functions; affiliates need not have a common name. It is possible but not necessary to introduce explicit subtypes such as bedroom and kitchen. Affiliate functions act like intelligent agents and apply only as needed. Thus, the abstract clean operation defined above would work as well on efficiency apartments, say, as on bedrooms and kitchens.

In effect, COO raises polymorphism to a higher level. Instead of just resolving overloading at runtime, cognitive (abstract) operations can incorporate any and all semantically related functions, irrespective of their names. Conceptually, abstract operations operate on any and all objects associated with a class hierarchy, irrespective of type. They can be defined to operate on any object where the operation makes sense semantically. In COO programming (as well as in design) one speaks of driving, cleaning or destroying a bicycle, car, train or airplane -- not of the latter being drivable, cleanable and destroyable.

In this sense, the precedence of objects and operations in COO programming may be reversed. Abstract operations act on sets of "real world" objects. They are not just virtual, however. Affiliate functions make them fully operational. Affiliates serve to modify what (sub-structures) abstract operations operate on. In this sense, affiliates (within objects) play a role roughly analogous to adjectives.

Data Execution.-- As noted above, data execution functions define relationships between objects (e.g., "done"). They also define relations between parent and children data elements (e.g., "room" and "bed" and "carpet"). This important construct is directly supported in the HLD language. In JAVA it is only enabled.

Thus, in COO programming we have:

```
[room] status (bed, carpet)
  [bed]: made
  [carpet]: dirty
clean (room)
```

The same thing can be done in defining data structures. In HLD we have:

```
[bed_room] construct (room, bed, carpet)
  [room]
  [bed]
  [carpet]
clean (bedroom)
```

where bedrooms are constructed from (ordinary) rooms, beds and carpets. In this case "construct" might simply add a bed and a carpet to the room (as is the
norm in inheritance), or it might involve more esoteric operations.

In JAVA (or as an alternative in HLD) we have, respectively:

\[ \text{clean (status (bed, carpet))} \]

and

\[ \text{clean (construct (room, bed, carpet))} \]

The former approach has the effect of replacing procedural complexity with structural complexity (cf. Scandura, 1981). This improves understandability in two ways: (a) it simplifies procedural statements and (b) it localizes data definitions.

Abstract HLD Operations.-- As noted above, abstract HLD operations go beyond simple polymorphism. In addition to allowing parameters to vary as to data structure, abstract operations, together with their affiliates perform the necessary behavior without specifying any additional procedural logic.

Consider the hybrid class/data hierarchy

\[
\text{[room]}
\text{[bedroom]}
\text{[bed]}
\text{[carpet]}
\text{[kitchen]}
\]

where \textit{bedroom} and \textit{kitchen} are subcategories of \textit{room}, and \textit{bed} and \textit{carpet} are components of \textit{bedroom}.

In HLD, without any additional work, we have:

\[ \text{clean (room)} \]

irrespective of whether the \textit{room} is a \textit{bedroom} or a \textit{kitchen}, or for that matter any hybrid type (e.g., efficiency apartment) in between.

In JAVA, in addition to defining a room class hierarchy, we would have to write and maintain clean functions for the bedroom and kitchen classes as well as for the room class. In the case of \textit{bedroom}, for example, the clean function might call \textit{make} and \textit{vacuum} in the bedroom class plus clean in the room class. Another clean would be needed for the kitchen class. An alternative in JAVA (or HLD) might be:

\[ \text{room.clean (room)} \]
\[ \text{bedroom.make (bed)} \]
\[ \text{bedroom.vacuum (carpet)} \]

The disadvantage of the first (most common) approach is that: (a) we have more than one \textit{clean} to maintain and/or to distinguish when debugging and (b) every new subclass (e.g., efficiency apartment) must be explicitly defined. The approach also tends to camouflage the semantic equivalence of \textit{make} and \textit{vacuum} with respect to \textit{clean}. The latter implementation is more explicit and provides more flexibility, but it adds procedural complexity, which can easily lead to sloppy programs that are hard to maintain.

Further simplification derives from the fact that data elements in HLD are typeless. All structures are explicitly represented hierarchically. This encourages the use of abstract objects (variables) in COO designs and programs. (The use of global variables is strongly discouraged with obvious advantages as regards system reliability, availability and maintainability.) Because
parameters may be as abstract as necessary, one is never forced into using long parameter lists. In addition, the core HLD language explicitly includes operations that work on arbitrary data structures, thereby facilitating modeling real-world behavior. The HLD operation

\texttt{duplicate (structure\_root, target\_element)}

for example, works on arbitrary node structures (such as those shown in Figures 1-4). In addition, basic HLD operations, such as \texttt{“add”, “:=”, etc.} are implemented (in C++) so they generalize across data types (e.g., int, float, char). Indeed, all additions to any HLD library are wrapped in \texttt{“node\_ptrs”} to ensure that they are typeless and interoperable. Compound operations constructed from these operations also interoperate based solely on real-world semantics.

\textit{Application of the Cognitive OO Programming Paradigm.---} We have developed, tested and refined a significant number of complex COO programs written in the extensible HLD language. Many of these programs are of the higher order variety and are quite powerful. That is, they operate on and modify (or build) new programs. These higher order programs perform tasks ranging from constructing call hierarchies and reorganizing modules in large C or Ada programs to translating C or FORTRAN into \textquote{good} Ada (e.g., Scandura, 1994). Currently available COO programs, however, were built using more or less standard top-down methods. To date, full use of the cognitive OO paradigm has been limited. Indeed, systematic design debugging at successive levels of analysis was not possible until very recently, among other things because the AutoBuilder Simulator did not fully support the simulation of aliases (pointers).

Because all HLD operations are implemented using \texttt{node\_ptrs} to \textquote{real world} objects, complications due to type differences are eliminated. COO programming is a natural outgrowth of COO design and simplifies program representation. Not only do designs map directly into the real world but so do core HLD operations. People can describe systems during implementation as well as during design in the way they naturally think about what needs to be done -- whether these abstractions refer to objects or to actions on those objects. Programs are described the way we speak and write -- using verbs for actions and nouns for objects. Similar thinking may also have been a prime motivation for languages such as COBOL. At that time, of course, modularization and the importance of structure were largely unknown, not to mention the possibility of encapsulating behavior within data structures. Characteristics of the COO paradigm are summarized in Table 1.

\textbf{Reuse of Legacy Code}

The COO paradigm, especially when coupled with design debugging provides a natural basis for reusing legacy code. Virtual components used in abstract models of application domains often map naturally onto executable components extracted from legacy code in those domains. Extracted code might (first) be converted into a modern language like JAVA or Ada (e.g., Scandura, 1994), or simply wrapped.
Code Extraction and Semantic Wrapping.-- To enable reuse in HLD, it is essential that legacy components be assessible by parameter passing -- allowing direct, clean mappings between high level designs and the to be reused components. This can be accomplished by encapsulating potentially reusable routines in semantic wrappers. Semantic wrappers replace language-specific data types with semantically meaningful relationships between data. Specifically, each component is wrapped in a node_ptr shell. (Note: Depending on intended semantics, semantically wrapped functions in C++ classes may either be called as is or affiliated with functions in other classes.)

Availability of needed domain-specific components (once semantically wrapped) makes it possible to implement arbitrary models in a way that directly reflects real world semantics. This clearly increases system maintainability. Furthermore, technical complications pertaining to data types and complex syntax, as well as initialization and complex interface issues, are eliminated. Semantic “glue” enabling wrapped components (from any source) to work together is incorporated directly within individual components. Once wrapped, it makes no difference whether the components originally came from existing executables (e.g., purchased libraries) or reusable legacy code.

The availability of core HLD components for manipulating real world data structures further facilitates the reuse of existing components. Even extraction processes can be facilitated using customizable reengineering and conversion components, which automatically gather information and/or otherwise improve legacy code (e.g., see Flexsys at www.scandura.com).

Semantics-Based (COO) Design.-- AutoBuilder’ visual (Flexform) Designer and Simulator environments make it possible to create, test and debug high level semantic-based models of arbitrarily complex, large scale software applications. Semantic core components support the interoperation of domain-specific components. Referenced

Table 1: General Characteristics of the Cognitive OO (COO) Paradigm

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Support for both “real world” objects and operations on those objects simplifies program representation. Systems are described like people think -- whether abstractions refer to nouns (i.e., objects) or verbs (i.e., operations).</td>
</tr>
<tr>
<td>(2)</td>
<td>Hierarchical relationships between objects are a natural consequence of process refinement and vice versa.</td>
</tr>
<tr>
<td>(3)</td>
<td>From a cognitive perspective, operations mapping real world input objects onto output objects correspond to conscious cognition. Functions defined between hierarchically related objects correspond in cognition to automatic (e.g., perceptual) processes -- automatic actions not requiring conscious thought. Functions between objects give meaning to those objects -- like “adjectives” modifying “nouns.”</td>
</tr>
<tr>
<td>(4)</td>
<td>There is no natural distinction between COO high and low level design. All distinctions are arbitrary and simply a matter of degree. The same is true of COO programming in the HLD language. The HLD language can be viewed as a preprocessor for C/C++ -- just as C++ was originally a preprocessor for C.</td>
</tr>
<tr>
<td>(5)</td>
<td>COO programs also can be implemented directly in C++ although the resulting syntax is often much more complicated. Generalizations of inheritance and evaluation, which arise naturally in COO programming, may be implemented in C++, albeit by resorting to more complex constructions. Abstract COO operations may be implemented by aggregating overloaded C++ functions, by introducing typeless node_ptrs as parameters (i.e., by wrapping) or by affiliating functions in class hierarchies with the abstract operations.</td>
</tr>
</tbody>
</table>
| (6)           | The HLD language supports higher order programming in the COO paradigm. These higher order operations add “adverbs” to the real world modeling facilities.
components may range in size from highly specialized routines to complete applications.

The above process can be used in developing any application. These applications can range from enterprise-wide systems (e.g., consisting of existing applications) to domain-specific application development systems for quickly constructing custom applications from wrapped components.

Applications built in HLD are not tied to AutoBuilder' development environment. Once a new application runs as it should in the Simulator, full source code can be generated automatically. Currently, AutoBuilder “HLD Generator” and “Semantic Integrator” support the C/C++ language. Other languages could easily be added or supported via bindings.

Semantic Wrapping.-- Although details are beyond the scope of this article, some progress also has been made in automating the semantic wrapping process. We have partially automated certain processes for standard C libraries, including reverse engineered (legacy) code. In addition, solutions have been designed for C++ classes and for integrating legacy source code in other languages such as Ada. Wrapping common middleware (e.g., for distributed computing), OLE and CORBA compliant objects, Motif widgets and integration with event driven code generated by modern GUI builders remains to be done.

Application to a Large C System. -- Although COO technology has already been used to develop real industrial strength programs (see below), broad-based application of higher order cognitive programming remains for the future.

Nonetheless, COO has been successfully applied in re-engineering a large C system. Specifically, cognitive methodology and supporting reengineering technology were used to completely redesign "top level" code in the PRODOC software engineering system. In the process, a large number of C components were extracted (from a system of 500K LOC), semantically wrapped and made interoperable. None of this work directly effected functionality of the system. Initially, the system did exactly what it had done before. From a maintenance (and enhancement) standpoint, however, the differences were dramatic.

Before redesign, the highest-level compilation unit in the system consisted of 72 C Flexform modules. After the redesign, all of the design logic was contained in only five HLD Flexforms. These Flexforms call about 65 semantically wrapped HLD Flexforms comprising the bulk of the lower level code. The latter are referenced resources but otherwise may be viewed as "black boxes".

As a result of the top-level redesign, the five design Flexforms are the only ones which need to be changed. Moreover, all of the semantically wrapped components are interoperable, not only with each other but also with every other semantically wrapped component. As a result, we have observed a dramatic reduction in the effort required to maintain, enhance and otherwise modify overall system logic. In this regard, length alone was reduced by a factor of 12 or more. (In terms of maintainability, one can make a case for complexity increasing by the square of the length -- i.e., a factor of 144.) Maintainability was
further enhanced by the dramatically simplified syntax.

Conclusions

The COO paradigm represents a potentially important advance in modeling and building software, offering a more complete representation of reality than does the OO paradigm. Among other things, COO design and programming allow more natural "real world" modeling and make it easier to test and debug systems. This in turn helps to simplify design and/or reduce complications characteristic of OO programming.

On the pragmatic side, COO programming can be implemented in C++. The generation of compilable C++ source code from COO designs is currently available and fully automatic. The only significant change COO requires is the increased design discipline it supports. This discipline has been shown to significantly reduce integration problems and improve maintainability. Nonetheless, the above represents only a beginning. COO design derives from a long history of research in structural (cognitive task) analysis. Although COO has been used successfully in informal HLD development, and in redesigning and reengineering a large (500K LOC) C system, this is still a work in progress. Highest priority in COO research at present is the separation of specification and design processes and automated support for the construction of internally consistent designs that are correct with respect to specifications. Another priority area is extending COO to accommodate interactions between objects.

References


Scandura, J.M. *Problem solving in schools and beyond: Transitions from the naive to the neophyte to the master.* *Educational Psychologist*, 1981, 16, 139-150.


