A method for increasing software productivity called object-oriented design—with applications for AI

by DAVID C. RINE

George Mason University
Fairfax, Virginia

ABSTRACT

Object-oriented design language research has suggested some basic concepts that object-oriented programming and languages should support. These are: (1) information hiding, (2) data abstraction, (3) dynamic binding, and (4) inheritance. Object-oriented languages are receiving extensive use in artificial intelligence. Although the Ada language possesses the information hiding and data abstraction concepts, it does not possess the dynamic binding and inheritance concepts. These and other limiting factors in developing AI software using object-oriented methods are discussed in this paper. Object-oriented design is becoming an important method for establishing database and knowledge base systems software as productivity issues rely more on tools for reconfiguring existing software and rapidly prototyping software under development. We present some features of object-oriented design pertaining to the development of such databases and knowledge bases including data modeling, data sublanguages, and distribution techniques.
INTRODUCTION: OBJECT-ORIENTED DESIGN LANGUAGES

Software development studies generally have confirmed that software development costs increase and software productivity decreases in more than a linear relationship as the size of the entire project increases.\(^1\) Software productivity also seems to depend upon such things as:

a. The amount of code.
b. The number of concepts that must be understood to make one programmer's subsystem interact properly with another's.

Software metrics have been developed in an attempt to measure software productivity that includes (a) and (b). Further, (b) is influenced by such concepts as:

c. Information hiding.
d. Data abstraction.

It has been observed that, consequently, one can reduce the amount of code written in some projects by acquiring most of the code parts from libraries containing pre-packaged code. Therefore, in this paper we do not think of databases or knowledge bases in the usual logical record-oriented fashion. We think of data banks of machine components—potentially live data banks that can be thought of as organized nests or hives of clones. These machine component banks, which we identify with logical entities termed objects, can be thought of as being managed in ways that are both similar and different from the ways logical records are managed. Ideally, the objects themselves should be able to adapt, similar to self-adaptive automata, depending on the context in which they are placed. Adaptation makes such banks ideally suited for rapid prototyping and simulating of artificial intelligence (AI) software systems. With a little effort, one can see that a language supporting the definition and manipulation of such banks, or bases, goes far beyond the generic language properties of Ada and into languages that possess powerful dynamic binding, inheritance, and adaptive properties.

Pascoe\(^3\) has suggested some basic concepts of object-oriented programming and languages. These are: (1) information hiding, (2) data abstraction, (3) dynamic binding, and (4) inheritance. OOLs are used a great deal in AI. Although the Ada language possesses concepts (1) and (2), it does not possess (3) and (4). Therefore, it may be asked to what extent do these concepts limit development of AI software?

In *Artificial Intelligence*,\(^4\) Winston suggests three approaches to answering the question: “Where is knowledge about procedures stored?” and later suggests that any of the three can be used in controlling a robot. Let us repeat these three approaches:

- A system exhibits action-centered control when the system's procedures know what subprocesses to use to perform actions.
- A system exhibits object-centered control when the system's class descriptions specify how to deal with objects in their own class.
- A system exhibits request-centered control when the system's procedures know their own purpose so that they may respond to requests.\(^5\)

In this paper, we address the second AI approach, object-centered control.

NEEDS: OBJECT-ORIENTED DESIGN METHODS IN AI

Rapid prototyping, simulating, and reconfiguring of systems are important to the development of AI software. In the past, managers of large organizations were only infrequently able to get answers to "what-if" questions. The reasons for this included:

- Rapid prototyping of such systems was non-productive.
- Simulating of such organizations were too costly to develop.
- Reconfiguring existing systems was not well-supported.

Also, management would attempt to take advantage of the experience of a systems analyst who had worked on similar cases to gain answers to "what-if" questions. More recently, however, we have begun to see the emergence of AI techniques\(^6\) to model large organizations. Such techniques are used to automatically generate the necessary scenarios for a particular business environment through rapid prototyping that supports rapid reconfigurability of potential systems.

Object-oriented programming is both a packaging technology and a software engineering method that addresses these software productivity issues. The kind of packaging approaches used will influence:

e. Software reconfigurability.
f. Ability to prototype rapidly.
g. The types of applications that may be developed.

Issues of object-oriented languages (OOL) began with the Small-talk-80 system (trademark of Xerox Corporation).\(^7\) The Smalltalk language offers a uniform and powerful metaphor
whereby procedures and data that belong together are packaged in an object. An object is a package of data and procedures that belong together, and Smalltalk procedures are called methods. An object can be thought of in many ways; for example, it can be casually thought of as a considerably expanded version of the Pascal record. Smalltalk does computation by sending messages to objects.

METHODS: DESIGN AND LANGUAGE

Abstract reasoning has played an important role in designing modern software. On the one hand, the role of structured systems analysis has made such tools as data flow diagrams and structure/HIPPO charts common in the top-down approach to procedure-oriented design. On the other hand, a bottom-up approach has been introduced by using existing packages and object-oriented designs. Shooman has suggested that certain applications are better developed through the top-down approaches, while other applications are more suited to bottom-up approaches. One factor, of course, is the amount of interaction anticipated between individual processes or modules. It has been suggested, for example, that the top-down approach is appropriate when there is a great deal of anticipated process/module interaction, and the bottom-up approach is more suitable when there is little, if any, interaction.

Booch has suggested an object-oriented design (OOD) method that includes: (1) defining the software engineering problem, (2) developing an informal strategy, (3) formalizing the strategy, and (4) implementing the solution. Formalizing the strategy includes: (a) identifying objects of interest by choosing them as nouns, pronouns, and noun clauses from the problem's text and (b) identifying operations of interest by choosing them as verbs, verb phrases, and predicates from the problem’s text. Moreover, it is pointed out that some objects identify classes of objects. Further, operations are identified to manipulate or act upon certain objects. It also is pointed out that only proper nouns and nouns of direct reference will represent objects at the code level, while other objects identify classes of objects. The method, therefore, implies an inheritance concept through the introduction of such classes.

It has been suggested that Ada is an OOL, but by our accepted definition, using concepts (1) through (4) described in the Introduction section, this is not the case. On the other hand, the OOD notation introduced in Booch's method, along with Ada, does imply the concepts of information hiding, data abstraction, and inheritance.

In summary, we are faced with a problem of mapping or transforming an OOD notation that has three of the properties of an OOL into a language, namely Ada, that has only two of the properties of an OOL. This problem may be severe because there is a possibility of losing information in the transformation.

The problem of losing information is similar to that faced by database designers when mapping entity-relationship (E-R) model diagrams, which clearly distinguish entities from attributes, into a relational model which may not. Lossless transformations are those for which it is guaranteed that information will not be lost. Identifying lossless transformations between design notations (language syntax) is a fundamental problem in systems analysis. Automating these transformations is somewhat like developing a language parser or reverse parser that will transform language “programs” into a lossless equivalent representation in object form (for example, where the object form is a relation, relation of relations, tree, or tree of trees).

Recent general research about AI software design using graphics support based upon hierarchies of data flow diagrams and knowledge base support using dictionaries of data definitions can be found in Harandi and Lubars.

DATABASE SUPPORT: SCHEMAS

AI software engineers develop material in high level “chunks” which may be thought of as software design schemas. Often such schemas are developed in an arbitrary manner and so the software designer must recall many rather detailed design “objects.” This further suggests a need for organized libraries of reusable code in software development tasks. In our framework, the high-level chunks are like database schemas. Design success may depend on the availability of schemas that logically locate desired library components and allow the software engineer to fit the components into the partially completed design.

Thus a library of these stored design schemas, which can be thought of as objects, and a system for schema manipulation are needed. These schemas, or objects, would be combined into an integrated knowledge base for use by a software engineer or program development expert system.

At the lowest logical level of detail in these schemas, one may casually think of certain objects as being like database records, each record being comprised of procedures, functions (modules), and data items; a notion similar to the most general kind of Ada record. These database records may also be thought of in terms of E-R relationships or relations in a relational model, but such that the entities can be like procedures and functions (modules) as well as data items. On the other hand, higher level objects evolve by use of superclasses from lower level classes, permitting the inheritance characteristics of object-oriented design.

Such schemas can be generated using object-oriented dictionaries as tools in the requirements gathering stage.

Moreover, messages and methods afford a means for generalizing the notion of data manipulation languages used in standard database management systems (DBMS).

Objects and messages in this context may be termed object sublanguages (borrowing from the notion of data sublanguage of DBMSs). An object base may also be managed in a distributed fashion. Tools for designing such systems essentially are the same as the partitioning algorithms used in setting up distributed databases.

Recent research has been carried out in this regard at a somewhat lower level of design abstraction that includes program structure as well as objects. In particular, Young has used the idea of a design template as an abstract and generic problem solution, which is applicable to a large number of...
such situations as we have mentioned here. Templates include
a generic procedural structure as well as an abstract defining
ability of data objects. Other results on relationships between
object-oriented design and database systems also have been
published.\textsuperscript{13}

**AI: OBJECT-ORIENTED DESIGN**

An early foundational work by P.J. Landin\textsuperscript{8} describes how
some of the semantics of Algol can be formalized by establish­
ing a correspondence between expressions of Algol and ex­pressions in a modified form of Church's lambda notation, an
important formalism of LISP. Landin describes a model for
computer languages and computer behavior that is based on
the notions of functional application and functional abstrac­tion. That model then is used as an abstract object language
into which Algol is mapped. The second part of Landin's
paper gives a formal description realizing an abstract compiler
into the abstract object language. Such mappings between
languages are an important part of system design.

The notion of "object" mentioned in this early paper is
similar to the modern notion of "object" mentioned herein,
since they are packaged parts of programs which can further be
built-up into classes, which in turn are objects.

Much AI software has been developed using LISP. More­
ever, relationships between Ada and LISP were reported at the 1985 AI-Ada Conference at George Mason University.

For example, there has been some work in developing LISP
translators in Ada, and LISP has been viewed as a higher-level
design tool for software that eventually will be coded in Ada.

The language ExperCommonLISP is one of the most com­prehensive OOLs for the Apple Macintosh because it imple­ments all the features of OOLs described in this paper except
unique instance methods.\textsuperscript{9} Classes, superclasses, and sub­classes, for example, are nicely implemented in this expansion
of CommonLISP. Another version of object-oriented LISP.
Zeta LISP, is available on the Symbolics AI workstations.

Therefore, from the computer language point of view, Ex­perCommonLISP incorporates more of the features of a true
OOL than does Ada. Hence, because of the need for lossless
transformations from design notation to computer language,
ExperCommonLISP may be a more desirable initial target
than Ada. This suggests a modification of the method of
OOD, geared toward Ada, introduced earlier.

The use of OOD also has appeared in PROLOG (another
popular AI language) language programming. Shapiro and
Takeuchi\textsuperscript{10} have observed that Concurrent PROLOG is ca­pable of expressing object-oriented language concepts, achiev­
ing the property of inheritance (i.e., the class-superclass hier­archy). In this approach certain goals can be thought of as
objects which accept messages. Presumably this observation
could be applied to (non-concurrent) PROLOG as well.

There is a correspondence between PROLOG goals and Ada
(also Pascal for that matter) procedures.

Advantages of an object-oriented approach to database sys­
tems design are described by Maiier and Stein.\textsuperscript{14} They state
that such an approach may result in a system that offers reduc­
tions in application development efforts beyond those achieve­able by traditional DBS approaches. Gem-\textsuperscript{Stone}\textsuperscript{13} is such an
object-oriented DBMS that affords packaging of both system
behavior and structure.

With data sublanguages and models of traditional DBMSs
such as those using the relational approach, a data model is
analogous to a fixed abstract data type which cannot change
over time as additional operators, for example, become im­portant to the application. Even when designing and im­plementing such a DBMS with a language having powerful data
abstraction and encapsulation capabilities (such as Ada) and
including the use of generics, it is not easy to change types and
expand the model (e.g., beyond the given relational model
implemented through Ada packaging). Therefore, the man­agement of changing types in an object-oriented database is
an important problem area.\textsuperscript{15}

**DISTRIBUTION OF OBJECTS:**

**PARTITIONING ALGORITHMS**

Previously, partitioning in database design has been a proce­
dure used to assign a logical object (e.g., relation in a rela­tion­al model) from a conceptual or external schema of the
database to one or more physical objects identified in internal
schemas (stored database). Further, in the design of a geo­graphically distributed database such logical objects (often
termed fragments) are assigned, with possible replication, to
the various geographical sites.

With such traditional databases Navathe, Ceri, Wieder­
hold, and Dou\textsuperscript{16} extended the work of Hoffer and Severance\textsuperscript{17}
by defining an algorithm in which attributes of an object are
permutated in such a way that attributes with "high affinity" are
clustered together. Further, information about the use of at­tributes by transactions is initially converted into a square
matrix, termed the attribute affinity matrix, a symmetric
square matrix \( u \) defined as follows:

\[
u_{ki} = \begin{cases} 
1 & \text{if transaction } k \text{ uses attribute } a_i \\
0 & \text{otherwise}
\end{cases}
\]

Their algorithm\textsuperscript{16} is a specialization of general algorithms
that permute rows and columns of a square matrix to obtain
a semiblock diagonal form, applied to partition a set of inter­acting variables into subsets which interact minimally.

In our context, the logical objects are objects as previously
defined, and transactions from users may be replaced by mes­sages from users or other objects.

Suppose that \( M_1, M_2, M_3, M_4 \) are messages or users
which refer to objects 01, 02, 03, 04, 05, 06. Then this can be
represented by the following incidence matrix \( MO = \)

<table>
<thead>
<tr>
<th>Objects</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M_1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( M_2 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( M_3 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( M_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Then if we break up the objects into the following groups it is possible to process the messages in parallel:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>03</td>
<td>06</td>
<td>05</td>
<td>02</td>
</tr>
<tr>
<td>04</td>
<td>05</td>
<td>06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another possible grouping follows, involving some duplication with fewer groups (it is no longer a partition):

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>03</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>05</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the grouping is carried out, it is possible to assign each group of objects to a segment of external storage—much like traditional program segmentation but at a higher level of abstraction—or to a node in a distributed system with its own processor and memory. Note that the processors may be non-von Neumann such as data flow processors.

In summary, the following factors must be considered:

1. Assignment of objects (and methods) to storage segments and nodes as well as the segment or node arrangement
2. Location of objects at storage segments or nodes and determination of all relevant addresses from user-supplied information and from information contained in a segment or node
3. Assumptions about whether all instances of an object are to be stored on a single storage segment or node (similar to horizontal partitioning).

**DATA FLOW: OBJECT-ORIENTED MODELING**

From the Introduction of this paper, recall that we referenced three ways of controlling a robot and have emphasized object-centered control. Each component in a computer-integrated manufacturing system may be a self-contained robot. The robots are therefore components of a distributed system. Bruno and Balsamo have described an object-oriented approach for modeling such systems using data flow concepts.

In this section, we take the position that data represents the internal state of a robot or automaton. Because of our interest in capturing the behavior and control characteristics of each object, robot or automaton, we will use the concept of a finite state machine (or process).

A finite state machine (fsm) is defined as a six-tuple \( (S_0, S, I, O, \delta, \lambda) \), \( S_0 \) in \( S \), \( S, I, O \) are finite sets, \( \delta \) and \( \lambda \) are functions, and they are related as follows:

- \( S_0 \)—the initial state of the fsm
- \( S \)—finite set of states (different data in memory)
- \( I \)—set of inputs
- \( O \)—set of outputs
- \( \delta: S \times I \rightarrow S \)—an input causes a state change
- \( \lambda: S \times I \rightarrow O \)—an input causes an output

With respect to the object-orientedness of the model, when a message \( m \) is sent to such an object an attempt is made to match the message with a selector \( i \) in \( I \) corresponding to a method of the object. If a match occurs, then the method is executed changing the internal state of the object and producing some output. If no match occurs, then the object does not change its state non-trivially.

It is also possible to consider another alternative when no match occurs. If message \( m \) does match with a selector corresponding to a method of the object, then a search may be made of those selectors in a fsm containing the given fsm as a subsystem, thereby allowing the object-oriented concept of supercing and inheritance.

**THEORY: CATEGORIES**

A category \( K \) comprises a collection \( OBJ(K) \), called the set of objects of \( K \), together with for each pair \( A, B \) of objects of \( K \) a distinct set \( K(A, B) \) called the set of morphisms from \( A \) to \( B \) subject to two conditions.

In the object-oriented design sense, an object \( A \) can be thought of as a "package" \( (S, P) \) pair of states (data) and processes (procedures) \( P \) that can receive messages \( f \) and send messages \( g \). When \( A = (S, P) \) receives a message corresponding to one of its methods associated with \( P \) it can change its internal state (manipulate its data accordingly). And data manipulation may include the sending of a message to another object \( B \).

An object category, \( OC \) can be formed from this object-oriented design concept by calling pairs \( (S, P) \) category objects, which are members of \( OBJ(OC) \), and by calling messages \( f (S_1, P_1) \rightarrow (S_2, P_2) \) category morphisms, which are members of \( MORPH(OC) \). It can be shown that this definition of \( OC \) satisfies the properties of a category.

An example of a pair \( (S, P) \) is an Ada package. However, since the Ada language does not possess the inheritance property of OOLs, subclasses and superclasses are not part of the language. Further in this respect, subobjects and superobjects do not occur naturally. Therefore, inclusion morphisms would not be a natural part of the corresponding category.
FUZZY SETS AND OBJECTS:
NATURAL LANGUAGES:

An object, we recall, is a package of data and method (operation) definitions. Associated with an object is its class, similar to the idea of instance of a class. For example, to say that “Pussy is a cat” is an abbreviated way of saying that “Pussy is an instance of class cat.” Thus, “Pussy” is a member of the class “cat.” Hence, we also have the idea of membership.

Using natural language, class instances can be purposefully quite abstract, often intentionally vague. For example, instead of saying “John is tall,” i.e., “John is a tall person,” we may say that “John is quite tall” or “John is very tall.” Quite tall is abstract, often intentionally vague. For example, instead of saying “John is tall,” i.e., “John is a tall person,” we may say “John is quite tall” or “John is very tall.” Quite tall and very tall are imprecise. Moreover, a fuzzy set in natural languages is a mapping from a set \( U \) into the closed unit interval of reals \([0,1]\). For example:

\[
\text{tall:} U \rightarrow [0,1]
\]

The range of tall determines the various grades or degrees of membership. Two attributes such as very tall or quite tall can be identified as similar or synonymous if there is sufficient overlap of their membership, as in common usage.

While fuzzy sets afford some measure of similarity, in the past it has been common in database design to consolidate these attributes as identical when they are sufficiently synonymous, sometimes creating a standardized word, such as tall, in order to remove any undesirable logical redundancies. This concept of fuzzy sets allows us to add the idea of degree of membership of an instance in a class, as well as the interesting notion of imprecise inheritance. Moreover, earlier we had interpreted an object as a stand-alone automaton that is capable of receiving and sending messages, as well as changing its internal states (data). In the context of this section a fuzzy object would be like a fuzzy automaton.

Fuzziness is an intrinsic property of natural language. This is one of many ways by which user-friendliness of software may be increased, including:

- Sentences
- Menus
- Levels of abstraction
- Mix of the above
- Approximate reasoning
- Syntax and semantics

In the second major step of database design and in the second step in knowledge base design, different initial user views are consolidated into a conceptual schema using the rules of identity, aggregation, and generalization. These rules can be thought of as class rules that rely upon “is a” to perform generalization and “is part of” to perform aggregation with identity classes that are synonymous, have similar semantic meaning, and have overlapping grades of membership such that their intersection is a basis for the identity. Moreover, aggregation and generalization allow for subclasses and superclasses based upon membership grades.

One of the important system components used to maintain fuzzy objects is a piece of software, known as a defuzzifier, between the user interface and knowledge base.

<table>
<thead>
<tr>
<th>TABLE I—Differences between Smalltalk-80 and Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding time</td>
</tr>
<tr>
<td>Operator overloading</td>
</tr>
<tr>
<td>Inheritance</td>
</tr>
<tr>
<td>Multiple inheritance</td>
</tr>
<tr>
<td>Classes</td>
</tr>
<tr>
<td>Information hiding</td>
</tr>
<tr>
<td>Data abstraction</td>
</tr>
</tbody>
</table>

SUMMARY: OBJECT-ORIENTEDNESS AND AI

Object-oriented programming is more a code packaging technique than it is a coding technique; and it is therefore a means by which software developers can encapsulate functional designs in a manageable fashion. While languages such as Smalltalk-80, LISP, PROLOG and Ada are very different languages, they do have certain object-oriented language properties in common which make each of them viable candidates for work in developing software for AI applications.

Table I summarizes some of the differences between Smalltalk-80 and Ada.

A class is sometimes referred to as a software integrated circuit in order to draw a comparison with the packaging of hardware silicon chips.

Further basic concepts of object-oriented programming may be found in Cox.

REFERENCES

13. Proceedings of the 1986 International Workshop on Object-Oriented Data-


