A MODEST SURVEY OF OBJECT-ORIENTED APPROACHES

David M. Bulman
Pragmatics, Inc.
Waikoloa, HI

Dave Bulman is president of Pragmatics, Inc., a consulting firm. He first programmed professionally in 1959. In 1962, he wrote the optimizing pass of one of the first Algol compilers. He has programmed in Algol, (Object) Pascal, Ada, 12 assembly languages and 10 other high-level languages. His responsibilities have ranged from developing real-time systems on microprocessors through managing database systems on mainframes to serving as chief of systems programming for a large on-line network.

A Modest Survey of OOD Approaches

David M. Bulman
President
Pragmatics, Inc.

The point of view taken in this article is that the two main problems in object-oriented design are: Deciding what objects are needed to solve a given problem and organizing them into a suitable architecture. Clearly, analysis of the requirements of the problem has to be done before the object-oriented design can be accomplished. In that sense, object-oriented design includes requirements analysis. A great deal has been published on objects, but the approaches to analysis and design appear to fit in a small number of categories.

INTUITION (I.E. "IT'S OBVIOUS").

Most of the earliest object-oriented programming advocates seemed to feel that deriving the objects was not a serious or interesting problem. For example, the first OOPSLA conference in 1986 had two papers on design notation, but none on how to do object-oriented design. The second OOPSLA in 1987 had just one paper on the subject; its author had written one of the papers on design notation in the first conference. There was, however, a workshop on the subject at OOPSLA '87. OOPSLA '88 also had exactly one paper on design. Suddenly, at OOPSLA '89, the problem seemed to be recognized; three or four papers were presented on design. During OOPSLA '90, object-oriented design (and analysis) seemed almost to jump out of the closet. Again, three or four papers were presented on the subject, and there was also a panel entitled "Structured Analysis and Object-Oriented Analysis." Unfortunately, essentially all the members of a later panel entitled "Reusability" agreed that the entire discussion in the earlier panel had represented completely outmoded ideas, since they agreed that object-oriented thinking could gain nothing from the earlier "structured" approaches.

A valuable book, Object-Oriented Analysis by Shlaer & Mellor [1] was published in 1988. In section 3.2 Identifying Objects, they said, "Identifying objects is pretty easy to do. . . . ask yourself, 'What are the things in this problem?'"
A MODEST SURVEY OF OBJECT-ORIENTED APPROACHES

David M. Bulman
Pragmatics, Inc.
Waikoloa, HI

Dave Bulman is president of Pragmatics, Inc., a consulting firm. He first programmed professionally in 1959. In 1962, he wrote the optimizing pass of one of the first Algol compilers. He has programmed in Algol, (Object) Pascal, Ada, 12 assembly languages and 10 other high-level languages. His responsibilities have ranged from developing real-time systems on microprocessors through managing database systems on mainframes to serving as chief of systems programming for a large on-line network.

The point of view taken in this article is that the two main problems in object-oriented design are: Deciding what objects are needed to solve a given problem and organizing them into a suitable architecture. Clearly, analysis of the requirements of the problem has to be done before the object-oriented design can be accomplished. In that sense, object-oriented design includes requirements analysis. A great deal has been published on objects, but the approaches to analysis and design appear to fit in a small number of categories.

INTUITION (I.E. "IT'S OBVIOUS").

Most of the earliest object-oriented programming advocates seemed to feel that deriving the objects was not a serious or interesting problem. For example, the first OOPSLA conference in 1986 had two papers on design notation, but none on how to do object-oriented design. The second OOPSLA in 1987 had just one paper on the subject; its author had written one of the papers on design notation in the first conference. There was, however, a workshop on the subject at OOPSLA '87. OOPSLA '88 also had exactly one paper on design. Suddenly, at OOPSLA '89, the problem seemed to be recognized; three or four papers were presented on design. During OOPSLA '90, object-oriented design (and analysis) seemed almost to jump out of the closet. Again, three or four papers were presented on the subject, and there was also a panel entitled “Structured Analysis and Object-Oriented Analysis.” Unfortunately, essentially all the members of a later panel entitled “Reusability” agreed that the entire discussion in the earlier panel had represented completely outmoded ideas, since they agreed that object-oriented thinking could gain nothing from the earlier “structured” approaches.

A valuable book, Object-Oriented Analysis by Shlaer & Mellor [1] was published in 1988. In section 3.2 Identifying Objects, they said, “Identifying objects is pretty easy to do. . . . ask yourself, ‘What are the things in this problem?’”
FROM A LARGE CLASS LIBRARY.

Many workers using a programming environment which includes a large library of classes indicate that most of the classes needed will be found in the class library [2, 3]. This approach has certainly been shown to work extremely well for problems where the existing class library exhibits a good match with the problem to be solved. There do not, however, seem to be any class libraries well-suited to the problem of, say, process control for a candy factory. Such libraries will probably be available sometime in the future, but not now.

If a large proportion of a problem involves providing an excellent user interface or can be cast into some of the familiar abstract data types such as stacks, queues, etc., then several programming environments include suitable class libraries. Such class libraries are also available from a growing number of third party vendors.

INHERITANCE.

This approach could be considered to be at least partly a subclass of the preceding approach: With a good enough class library, most required classes can simply be inherited from already existing classes. The number of papers and books espousing this idea are too numerous to cite here. Virtually every author working with a true object-oriented programming language with a powerful programming support environment appears to support this view. The same limitation applies here as with any normal dependence on a class library; the library must be well-suited to the problem at hand.

In addition, recognizing the opportunity for inheritance seems to work well when a small, highly-capable team is working on a problem they are familiar with, and which is small enough to be completed in time by such a small team. Otherwise, most opportunities for using inheritance seem to be discovered only after the programmer has implemented quite a few objects and recognizes that the commonality between them could be abstracted into a higher-level class.

Those who emphasize the importance of inheritance in object-oriented programming usually discuss class hierarchies in terms of top-down refinement of abstract classes. In my opinion, however, when solving significant problems in the real world, most members of inheritance hierarchies are discovered bottom-up. I would even be tempted to support the extreme statement that inheritance is more of an implementation tool than an analysis or design tool.

NOUNS & VERBS.

Another name for this approach is the “Paragraph Method.” It holds that you should write a paragraph in English (or any natural language) which describes the problem. Then you should look for the objects/classes among the noun phrases in this description. The most extreme proponents suggest underlining the nouns with one color ink and the verbs with another color. Some authors go on to say that many of the verb phrases in the description will represent the operations on the objects.

The first publication of this method described it by stating that you should write an informal description of the solution. This was especially troublesome to me because the solution of an object-oriented design would necessarily involve deciding what the objects in the solution should be, but the first step in the object-oriented design was to describe the solution. If this sounds confusing, it is because the process sounds circular to me: First decide what the object-oriented solution is; then decide what objects constitute that solution.

It is very difficult to argue against this method because its efficacy is usually proved using logic of the form: If A is true then B must also be true; B is true, therefore A. That is, some proponents will describe, in English, an object-oriented design which was derived by some other method. Then they underline the nouns and note that these are the objects in the design. This is an interesting, and possibly valuable, method of documentation. It is not, however, a design method.

One project at a large aerospace company using nouns and verbs was described in a public meeting a couple of years ago. Six competent people attempted to write paragraphs describing the problem, so that they could derive the objects from the nouns. It took them four weeks to arrive at the first paragraph they could all agree was suitable. It is clear to me that some of them had already decided on several of the objects before starting to write the paragraphs. The reason it took them so long to agree on the first paragraph was that they were trying to figure out an acceptable wording of the problem which would contain the nouns and verbs describing the objects they had derived by some other “method.”

Using nouns and verbs in object-oriented design was apparently first proposed by Abbott [4]. It’s popularity is largely due to the first edition of Booch’s first book [5]. It is interesting that one of the major changes in the second edition [6] deemphasizes this approach. Booch himself has often said that the method does not scale up to larger problems. His reason for using it in his book was that it is a good way for the beginner to start to understand objects.

The reason I am devoting so much space to the use of nouns and verbs is that several of the other approaches described here are little more than elaborate notational schemes. While some of the notations are well-developed, the actual methods associated with several of them boil down to examining a description of the problem for the important things mentioned in the requirements.

Thus, although the pure nouns and verbs method has been fairly completely discredited, traces of it appear in several other methods. Certainly many of the objects in an object-oriented design can be found in this way. Unfortunately, having found the obvious objects from the nouns, usually only vague guidance is available for the remaining objects.
FROM A LARGE CLASS LIBRARY.

Many workers using a programming environment which includes a large library of
classes indicate that most of the classes needed will be found in the class library [2, 3]. This
approach has certainly been shown to work extremely well for problems where the existing
class library exhibits a good match with the problem to be solved. There do not, however,
seem to be any class libraries well-suited to the problem of, say, process control for a candy
factory. Such libraries will probably be available sometime in the future, but not now.

If a large proportion of a problem involves providing an excellent user interface or
or can be cast into some of the familiar abstract data types such as stacks, queues, etc., then
several programming environments include suitable class libraries. Such class libraries are
also available from a growing number of third party vendors.

INHERITANCE.

This approach could be considered to be at least partly a subclass of the preceding
approach: With a good enough class library, most required classes can simply be inherited
from already existing classes. The number of papers and books espousing this idea are too
numerous to cite here. Virtually every author working with a true object-oriented program-
ing language with a powerful programming support environment appears to support this
view. The same limitation applies here as with any normal dependence on a class library; the
library must be well-suited to the problem at hand.

In addition, recognizing the opportunity for inheritance seems to work well when a
small, highly-capable team is working on a problem they are familiar with, and which is
small enough to be completed in time by such a small team. Otherwise, most opportunities
for using inheritance seem to be discovered only after the programmer has implemented quite
a few objects and recognizes that the commonality between them could be abstracted into a
higher-level class.

Those who emphasize the importance of inheritance in object-oriented programming
usually discuss class hierarchies in terms of top-down refinement of abstract classes. In my
opinion, however, when solving significant problems in the real world, most members of
inheritance hierarchies are discovered bottom-up. I would even be tempted to support the
extreme statement that inheritance is more of an implementation tool than an analysis or
design tool.

NOUNS & VERBS.

Another name for this approach is the "Paragraph Method." It holds that you should
write a paragraph in English (or any natural language) which describes the problem. Then
you should look for the objects/classes among the noun phrases in this description. The most
extreme proponents suggest underlining the nouns with one color ink and the verbs with
another color. Some authors go on to say that many of the verb phrases in the description
will represent the operations on the objects.

The first publication of this method described it by stating that you should write an
informal description of the solution. This was especially troublesome to me because the
solution of an object-oriented design would necessarily involve deciding what the objects in
the solution should be, but the first step in the object-oriented design was to describe the
solution. If this sounds confusing, it is because the process sounds circular to me: First
decide what the object-oriented solution is; then decide what objects constitute that solution.

It is very difficult to argue against this method because its efficacy is usually proved
using logic of the form: If A is true then B must also be true; B is true, therefore A. That is,
some proponents will describe, in English, an object-oriented design which was derived by
some other method. Then they underline the nouns and note that these are the objects in the
design. This is an interesting, and possibly valuable, method of documentation. It is not,
however, a design method.

One project at a large aerospace company using nouns and verbs was described in a
public meeting a couple of years ago. Six competent people attempted to write paragraphs
describing the problem, so that they could derive the objects from the nouns. It took them
four weeks to arrive at the first paragraph they could all agree was suitable. It is clear to me
that some of them had already decided on several of the objects before starting to write the
paragraphs. The reason it took them so long to agree on the first paragraph was that they
were trying to figure out an acceptable wording of the problem which would contain the
nouns and verbs describing the objects they had derived by some other "method."

Using nouns and verbs in object-oriented design was apparently first proposed by
Abbott [4]. It's popularity is largely due to the first edition of Booch's first book [5]. It is
interesting that one of the major changes in the second edition [6] deemphasizes this ap-
proach. Booch himself has often said that the method does not scale up to larger problems.
His reason for using it in his book was that it is a good way for the beginner to start to
understand objects.

The reason I am devoting so much space to the use of nouns and verbs is that several
of the other approaches described here are little more than elaborate notational schemes.
While some of the notations are well-developed, the actual methods associated with several
of them boil down to examining a description of the problem for the important things
mentioned in the requirements.

Thus, although the pure nouns and verbs method has been fairly completely
discredited, traces of it appear in several other methods. Certainly many of the objects in an
object-oriented design can be found in this way. Unfortunately, having found the obvious
objects from the nouns, usually only vague guidance is available for the remaining objects.
In English, as the linguists say, "any noun can be verbed." The converse, but slightly less pungent statement, "any verb can be nominalized," is also valid. Thus, the objects derived by dependence on noun phrases in the problem description can depend on the personal writing style of the requirements analyst.

Finally, most of the verbs in the description are probably not really sources of operations on the objects at all. It is far more likely that the verbs will represent relationships between entities in the information model of the problem.

SAME ENTITIES IN SOLUTION AS IN PROBLEM.

Bertrand Meyer in [7] (p. 326) says, "Perhaps the most useful technique for finding classes ... is to look for meaningful external objects. Many classes just describe the behavior of objects from the abstract or concrete reality being modeled ... . This idea often yields the fundamental classes of a system, obtained directly from their external counterparts."

Many very capable object-oriented designers approach problems in just this way. Meyer has described this approach more articulately than most other authors. It is certainly a suitable basis for deriving many of the objects in small problems and some of the objects in more complex problems.

INFORMATION MODELING.

Several of the best regarded methods are variations on the information modeling theme [1, 8-10]. Some of them use fairly classic entity-relationship diagrams, while others introduce various forms of "object-diagrams" which show objects and different kinds of relationships between them.

Each of these methods have added useful techniques for representing different aspects or "dimensions" to the model of the problem or its solution. I think some variation of these is almost certain to be used in any successful graphical approach.

In some sense, each of these pushes the big question back in the same way. How do you go about deciding which are the entities of interest? At least by drawing diagrams of the entities, your attention is focussed on them and their interrelationships. To see how to actually do information modeling, it is useful to go to books dealing specifically with the subject [11]. Don't be discouraged by the title of this or similar books. Part of it specifically treats the problem of logical data modeling. In each such discussion of information modeling, the first step always seems to boil down to the same advice: Either listen to your user or read the requirements statement. Then, look for major objects of interest.

OBJECT MODELING.

Object Modeling could be said to be the object-oriented design rendering of Information Modeling. A recent book [10] describes an approach the authors call the Object Modeling Technique. As is the case with several other authors, they find it useful to partition their model into several different viewpoints. They call their three viewpoints the Object Model, the Dynamic Model, and the Functional Model. The first step in the method is the same as in most of the other methods: Identify the objects or classes. Their advice is "As shown in Figure 8.4, begin by listing candidate object classes found in the written description of the problem. Don't be too selective; write down every class that comes to mind. Classes often correspond to nouns."

DATA FLOW DIAGRAMS.

The question of whether or not a Data Flow Diagram (DFD) is useful for object-oriented design has attained almost the same religious status as which text editor or which programming language is best. Perhaps the most important thing to note about data flow diagrams is that they have little to do with functional decomposition. They describe the flow of data in a system, where it is stored, and how it is communicated to and from the outside world. What some people refer to as the functions in a DFD were described by their originator as "rules for transforming the input data to the output data."

In particular, DFDs can be used to discover objects which represent stores of data describing things in the problem domain. They can also be used to find objects which encapsulate interfaces between the system and the outside world. Descriptions of the operations for the objects are found in the data transforms which access the objects found in the DFDs.

RESPONSIBILITIES.

A relatively new approach to object-oriented design could be called "responsibility driven" design [12]. Instead of attempting to define the operations directly, an intermediate step is useful. First describe the responsibilities of each object, without being too detailed in the description. Later derive the required operations from the responsibilities. This has the advantage of dividing one large step into two smaller ones, making the task easier to conquer.

REFERENCES


In English, as the linguists say, "any noun can be verbed." The converse, but slightly less pungent statement, "any verb can be nominalized," is also valid. Thus, the objects derived by dependence on noun phrases in the problem description can depend on the personal writing style of the requirements analyst.

Finally, most of the verbs in the description are probably not really sources of operations on the objects at all. It is far more likely that the verbs will represent relationships between entities in the information model of the problem.

SAME ENTITIES IN SOLUTION AS IN PROBLEM.

Bertrand Meyer in [7] (p. 326) says, "Perhaps the most useful technique for finding classes ... is to look for meaningful external objects. Many classes just describe the behavior of objects from the abstract or concrete reality being modeled . . . . This idea often yields the fundamental classes of a system, obtained directly from their external counterparts."

Many very capable object-oriented designers approach problems in just this way. Meyer has described this approach more articulately than most other authors. It is certainly a suitable basis for deriving many of the objects in small problems and some of the objects in more complex problems.

INFORMATION MODELING.

Several of the best regarded methods are variations on the information modeling theme [1, 8-10]. Some of them use fairly classic entity-relationship diagrams, while others introduce various forms of "object-diagrams" which show objects and different kinds of relationships between them.

Each of these methods have added useful techniques for representing different aspects or "dimensions" to the model of the problem or its solution. I think some variation of these is almost certain to be used in any successful graphical approach.

In some sense, each of these pushes the big question back in the same way. How do you go about deciding which are the entities of interest? At least by drawing diagrams of the entities, your attention is focused on them and their interrelationships. To see how to actually do information modeling, it is useful to go to books dealing specifically with the subject [11]. Don't be discouraged by the title of this or similar books. Part of it specifically treats the problem of logical data modeling. In each such discussion of information modeling, the first step always seems to boil down to the same advice: Either listen to your user or read the requirements statement. Then, look for major objects of interest.

REFERENCE.


DERIVING OBJECTS: IS IT DIFFERENT FOR EMBEDDED SYSTEMS

David M. Bulman
Pragmatics, Inc.
Waikoloa, HI

Dave Bulman is president of Pragmatics, Inc., a consulting firm. He first programmed professionally in 1959. In 1962, he wrote the optimizing pass of one of the first Algol compilers. He has programmed in Algol, (Object) Pascal, Ada, 12 assembly languages and 10 other high-level languages. His responsibilities have ranged from developing real-time systems on microprocessors through managing database systems on mainframes to serving as chief of systems programming for a large on-line network.