Object-Oriented Design: Deriving Conceptual Solutions to Large- Scale Information Processing Problems

M. A. Whiting

May 1990

**Prepared for the U.S. Department of Energy
under Contract DE-AC06-76RLO 1830**

**Pacific Northwest Laboratory
Operated for the U.S. Department of Energy
by Battelle Memorial Institute**

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.** Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government of any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST LABORATORY
operated by
BATTELLE MEMORIAL INSTITUTE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC06-76RLO 1830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831;
prices available from (615) 576-8401. FTS 626-8401.

Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

NTIS Price Codes, Microfiche A01

Printed Copy

Price Code	Page Range	Price Code	Page Range
A02	1- 10	A15	326-350
A03	11- 50	A16	351-375
A04	51- 75	A17	376-400
A05	76-100	A18	401-425
A06	101-125	A19	426-450
A07	126-150	A20	451-475
A08	151-175	A21	476-500
A09	176-200	A22	501-525
A10	201-225	A23	526-550
A11	226-250	A24	551-575
A12	251-275	A25	576-600
A13	276-300	A99	601-Up
A14	301-325		

**OBJECT-ORIENTED DESIGN:
DERIVING CONCEPTUAL SOLUTIONS TO LARGE-
SCALE INFORMATION PROCESSING PROBLEMS**

M. A. Whiting

May 1990

Prepared for
the U.S. Department of Energy
under Contract DE-AC06-76RLO 1830

Pacific Northwest Laboratory
Richland, Washington 99352

PREFACE

The Vertical Integration of Science, Technology, and Applications (VISTA) Project is a long-term effort conducted by the Pacific Northwest Laboratory (PNL)^(a) directed toward accelerating the process of making research results (data, models, advanced concepts) usable and available to R&D applications. The initial goal of the program is to develop a software-based information system to guide the assessment and remediation process for hazardous waste sites at the U.S. Department of Energy (DOE) facilities. The information

system will link users (DOE, laboratories, and remediation contractors) to computer models and technical data available at PNL, to speed up the remediation process, while decreasing costs and accelerating the deployment of new technologies.

This report describes a methodology used to design components of the VISTA information system based on an object-oriented computing model.

(a) Operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

ACKNOWLEDGMENTS

The object-oriented design methodology described in this report evolved through the cooperative work of the VISTA System Architecture Group. The team members who

participated in the formulation of this methodology include Janice R. Lewis, Edward Kleckner, and Marcie Olander.

SUMMARY

Object-oriented design (OOD) is a new methodology for studying, planning, specifying, and documenting solutions to information processing problems. OOD embodies aspects of the object-oriented programming philosophy (which is currently being applied to the *implementation* of software-based solutions) to provide enhanced tools and techniques for design. The fundamental difference between OOD and conventional design methodologies is that OOD integrates consideration of data and procedure so that they are viewed in balance throughout the design process. Traditionally, emphasis in design has been on procedural specification alone or directed toward data and process independently.

The VISTA project team at Pacific Northwest Laboratory has adopted OOD for use in the conceptual design of VISTA system components. This approach was used because traditional design procedures (e.g., top-down functional decomposition) did not produce acceptable results when applied to our design problems.

The successful application of the methodology relies on the use of objects to define the solution. An object is a black-box entity capable of performing pre-defined operations in response to messages. The use of objects inherently provides for modularity and information hiding, both important to the construction of dependable, reusable software. Another powerful mechanism associated with the use of objects is inheritance. Inherited operations and data from generic objects provide greater potential for reusability in descendant, specialized objects.

The OOD process essentially consists of three phases:

- define the problem
- construct an informal solution strategy
- create the object-oriented design blueprint.

The problem is defined by specifying a subset of the user requirements to be investigated. Construction of the informal solution strategy results simply in an English language statement of the design team's proposed solution to the stated problem. Once the design team (in concert with the user group) has decided that they have accurately captured a problem, and the solution (if implemented as stated) would solve the problem, they may then move into the more formal design phase.

Creating the object-oriented design blueprint consists of a top-down decomposition of the objects that comprise the solution space. The process begins by identifying the objects and their operations at the highest conceptual level of the solution. Object attributes are identified and interfaces are specified. Relationships between objects (e.g., object A depends on object B to perform its operation) are specified. When this is complete, component parts of the high-level objects are identified as new objects to be investigated, and the designer iterates back to identify their operations, attributes, and relationships. Iteration continues until the solution space has been sufficiently defined to satisfy the design team's requirements.

The result of this design process is a conceptual design specification. This specification is then handed off to low-level designers or programmers who create a detailed design specification or implementation description. Artifacts of the conceptual design specification include: object design specifications, object-relationship diagrams, and state-transition diagrams (object life cycles). Throughout this process it is highly recommended that the design team stay in close contact with the user group and technical peers to ensure that the design continues to respond to user requirements. Therefore, scenario descriptions and potential user views may be included in the conceptual design.

The OOD process has worked well to date in the design of VISTA system components. In the same manner as object-oriented programming languages allow programmers freedom to abstract low-level solutions to a greater extent than conventional languages

allow, OOD allows designers greater freedom in high-level design expressions. The results of OOD greatly contribute to the longer term goals of creating dependable, reusable software for the VISTA project.

CONTENTS

PREFACE	iii
ACKNOWLEDGMENTS	v
SUMMARY	vii
1.0 INTRODUCTION	1
2.0 BACKGROUND	3
2.1 MAINSTREAM LANGUAGE EVOLUTION AND DESIGN	3
2.2 NON-MAINSTREAM LANGUAGE EVOLUTION AND DESIGN	3
3.0 OBJECT-ORIENTED CONCEPTS	5
3.1 OBJECTS AND MESSAGES	5
3.2 CLASSES AND INHERITANCE	7
4.0 OBJECT-ORIENTED DESIGN METHODOLOGY	9
4.1 PREPARATION AND GOAL SETTING	9
4.2 PROBLEM STATEMENT	10
4.3 INFORMAL SPECIFICATION	10
4.3.1 Specify an Informal Solution	10
4.3.2 Informally Review Problem and Solution	10
4.3.3 Refine the Problem Statement and Informal Specification	10
4.4 OBJECT-ORIENTED BLUEPRINT	11
4.4.1 Identify Top-Level Conceptual Objects in the Solution Space	11
4.4.2 Identify Object Relationships and Object Dependencies	11
4.4.3 Test Solution with Scenarios	11
4.4.4 Iterate Until Level of Design Has Reached Appropriate Depth	12
4.5 DETAILED DESIGN - HAND OFF	12
5.0 TOOLS FOR OBJECT-ORIENTED DESIGN	13
6.0 CONCLUSIONS	17

7.0 REFERENCES.	19
APPENDIX - OBJECT-ORIENTED DESIGN EXAMPLE.	A.1

FIGURES

1	Object-Message Mechanism.	6
2	Inheritance Hierarchy	6
3	Object Design Template	14
4	Object-Relationship Diagram	14
5	Booch Interface Diagram.	15

1.0 INTRODUCTION

Object-oriented design (OOD) is a new methodology for studying, planning, specifying, and documenting solutions to information processing problems. OOD embodies aspects of the object-oriented programming philosophy (which is currently being applied to the *implementation* of software-based solutions) to provide enhanced tools and techniques for design. The fundamental difference between OOD and conventional design methodologies is that it integrates consideration of data and procedure so that they are viewed in balance throughout the design process. Traditionally, emphasis in design has been on procedural specification alone or directed toward data and process independently.

Object-oriented design inherits three notable characteristics from object-oriented programming: abstraction, information hiding, and modularity. Enhanced *abstraction* capabilities means that real world entities, operations, and relationships may be modelled as they are discovered by designers while considering solution possibilities. Previously, designer thinking was strongly influenced by the limitations of the available programming languages for representing data and procedure. *Information hiding* enforces the philosophy that a client requiring the services of an object need only see the object's interface. That is, a client needs to know only *what* services an object can provide, and not *how* it accomplishes the service. This protects both the client's security in relying upon a service to be provided as well as the supplying object's integrity in providing only what is specified. *Modularity* refers to the compartmentalization of data and procedure into logical, manageable units. It is encouraged through the conventions of object definition and through a property of object systems called *inheritance*. Inheritance of operations and data through object hierarchies is discussed in later sections of this paper. Object-oriented design methodology is aptly commented upon by Wiener and Sincovec (1984):

No longer is it necessary for the system designer to map the problem domain into

predefined data and control structures present in the implementation language. Instead the designer may create his or her own abstract data types and functional abstractions and map the real world domain into these programmer-created abstractions. This mapping, incidentally, may be much more natural because of the virtually unlimited range of abstract types that can be invented by the designer. Furthermore, software design becomes decoupled from the representation details of the data objects used in the system. These representation details may be changed many times without any fallout efforts being induced in the overall software system.

Pacific Northwest Laboratory uses OOD to design solutions to information processing problems being addressed by the Vertical Integration of Science, Technology and Applications (VISTA) project. The VISTA team has chosen the OOD methodology, even though its progenitor, object-oriented programming, is still in its infancy. Object-oriented programming is being used on a limited, but increasing, number of software implementations, few of which are of significant size (Stroustrup 1989). OOD procedures are being used on even fewer projects. The techniques are neither completely formalized, nor standardized. Successful achievement of certain stages of the object-oriented methodology described in this document is dependent upon the design team determining their own satisfactory procedures for accomplishing the outlined goals for these stages. This implies a certain degree of experimentation. Yet, for the VISTA project to date, the benefits of the methodology as a whole have outweighed the difficulties the non-standardized areas have presented. In fact, this methodology has allowed the team to overcome many difficulties we would have encountered had we chosen to pursue conventional design approaches.

This report is sub-titled "Deriving Conceptual Solutions to Large-Scale Information Pro-

cessing Problems" because it focuses on capturing, performing, and managing the conceptual design process. At a later date, a paper is planned to discuss how to derive implementation descriptions using object-oriented methodologies. The results of conceptual design provide the basic input to the implementation (or detailed) design process, yet, in our experiences at PNL, we have found these two processes are most efficiently achieved using

different approaches (essentially following the maxim, "conceptually design from top-down; design in detail from the bottom-up").

The introduction to this paper is followed by sections describing the background of language evolution and design methodologies, basic object-oriented concepts, the object-oriented design methodology, and conclusions.

2.0 BACKGROUND

This section briefly reviews the evolution of programming languages and design methodologies that lead up to the development of OOD.

2.1 MAINSTREAM LANGUAGE EVOLUTION AND DESIGN

The evolution of conventional programming design methodologies corresponds to some degree with the evolution in programming language capabilities. One measure of design adequacy is in the distance between the abstract design concepts and their resulting software representation. Generally, the closer the language representation comes to expressing the abstract design, the more certain the designers are that it accurately embodies the solution.

The first computer languages, assembly languages, left very little room for abstraction in data or procedure. Therefore, the distance between design concepts and their representations were quite high, and programmers had to take meticulous care in ensuring programs functioned as designed. The first high-level languages, FORTRAN and COBOL, permitted improved levels of procedural abstraction, yet entities in the real-world problem space had to be translated into a form compatible with the static, pre-defined data structures incorporated into the languages. Early (pre-1975) design methodologies placed heavy emphasis on procedural representation. Work in this area concentrated on the development of "modular programs" (Yourdon and Constantine 1979), and methods for "top-down" software refinement (Wirth 1971), eventually arriving upon the concept of "structured programming."

In the 1970s, the concepts of data abstraction and information hiding emerged in computer science, and data-driven design techniques

grew more popular with software engineers. Notable work in integrating data structure and data flow into the design definition was performed by Constantine (1974), Jackson (1975), and Warnier (1974). Programmers consistently emphasized procedural design throughout the '70s, however, until languages, such as Pascal, appeared accommodating these new data-related concepts. Pascal's program-mable data structures and types encouraged thinking about data abstraction in day-to-day programming.

2.2 NON-MAINSTREAM LANGUAGE EVOLUTION AND DESIGN

Data abstraction was emphasized in some languages as part of their fundamental design. These languages were not mainstream programming tools for various reasons including poor execution performance and entrenchment of the language within research laboratories. Simula (Dahl et al. 1967), built by Kristen Nygaard and Ole-Johan Dahl of the University of Oslo and the Norwegian Computing Center, is a general-purpose programming language that is an object-oriented extension to Algol-60. Objects, classes, inheritance, and later information hiding, many of the features being offered as innovative concepts in recent languages, were defined and used intrinsically in the Simula language. Smalltalk (Goldberg and Robson 1983), originated by Alan Kay, introduced an object-oriented language in its exemplary form. The language was developed at the Xerox Palo Alto Research Center as part of their work in creating an advanced personal workstation. Both Simula and Smalltalk pioneered radical departures from procedurally-oriented languages and formed the foundation for modern-day object-oriented programming.

3.0 OBJECT-ORIENTED CONCEPTS

The next three sections define object-oriented terminology and introduce object-oriented concepts to the reader to form the basis for understanding the object-oriented design concepts to follow.

3.1 OBJECTS AND MESSAGES

An *object* is a black-box entity capable of performing pre-defined operations in response to *messages*. It contains publicly-accessible and private data, and publicly-accessible and private operations. An object knows how to interpret messages sent to it and determine which operation to perform in response. Messages provide the interface to an object—they are the only mechanism with which to make an object perform a behavior (that is, execute an operation) or to respond to a request for data. Messages may either be imperative (a command whose response is some behavior by the object) or interrogative (a request for information or data to be returned to the message sender). An example of the object-message mechanism is shown in Figure 1.

All public access to objects occurs through the message interface. Messages are received across the interface and specify what operation (*method* in object-oriented parlance^(a)) the object should perform (in the sense that "what" the operation accomplishes is known but not "how" it is accomplished). The object receiving the message determines how the message should be handled. The message contains a symbolic name, called the *selector*, that describes the desired operation. If there is a *method definition* that corresponds to the method selector, the object performs the operation specified. The operation may involve returning the publicly-accessible data or executing publicly-accessible operations. Private data or private operations may be accessed by the object in order to accomplish its task, but external entities know nothing of this. The details of the method definition are hidden from the external entities. This characteristic is known as *information hiding*.

Objects, incorporating both data and operation, inherently provide *data abstraction* and *modularity*. Data abstraction is used to describe the coupling of data and operation within the object definition. The protection provided by the message interface and the information hiding is known as *encapsulation*. In a different sense, the grouping of data and operation with a well-defined interface provides modularity. Programmers naturally tend toward coding modular objects. Objects growing to unmanageable size are easily decomposed into groups of objects. Achieving modularity with encapsulation is important to the construction of dependable, reusable software.

3.2 CLASSES AND INHERITANCE

A *class*, by conventional definition, is a group of entities sharing common attributes. Objects are grouped in classes to provide for attribute sharing. For example, **myAddressBook** may be considered a member of the class of **AddressBook**.^(b) An object that has been created from a class is an *instance* of that class.

The class structure provides an opportunity for software reusability, that is not found in conventional programming languages, through a mechanism known as *inheritance*. For example, suppose **myAddressBook** is an instance of class **AddressBook**. **AddressBook** may be a specialization of a generic class **Book**. **AddressBook** is then called a *subclass* of **Book**. The difference between these two classes is in the characteristics of an address book—its information is organized around names, addresses, and telephone numbers. Inheritance allows data and operations to be passed down from **Book** to **AddressBook** and thereby be accessible to **myAddressBook**. Figure 2 diagrams this inheritance hierarchy. Reusability becomes highly notable when other subclasses are defined in consideration of existing classes. In Figure 2, **FictionalBook** and **Dictionary** are also

Figure 1. Object-Message Mechanism.

Figure 2. Inheritance Hierarchy.

created as specializations of class **Book**. Instances of both of these classes may use methods defined by **Book**.

Inheritance could be used for the **myAddressBook** object as follows. Suppose a method **findString** is defined as part of the class **Book**. Object **myAddressBook** may wish to use this method in fulfilling its method that defines **findPerson**. Inheritance allows the object to access this method

without having to rewrite or redefine the operation since **myAddressBook** is in the inheritance hierarchy of **Book**.

The use of classes and inheritance forms the basis of software reusability in object-oriented systems. Some researchers have envisioned large software component libraries being constructed of reusable, modifiable class definitions. Cox (1986) refers to these as "Software ICs."

(a) The object-oriented terminology used in this paper is derived from the Smalltalk language description, i.e., "method" as applied to objects originated with Smalltalk and may not be the same term used by others when describing procedural information.

(b) Again, the Smalltalk convention of naming objects and classes is used. Class names begin with an upper-case letter and object and method names begin with a lower-case letter. Extended names are concatenated and all succeeding words begin with an upper-case letter, for example, "objectName."

4.0 OBJECT-ORIENTED DESIGN METHODOLOGY

The object-oriented design philosophy separates the design phase of the software engineering lifecycle into two segments, as mentioned earlier. The first phase is one of conceptual design, where the solution space of the problem as a whole is developed. The second phase is that of detailed design, where an implementation description is derived. This separation has proven particularly useful for large-scale problems requiring a great amount of conceptual design work from multi-person design teams. The following sections describe the top-down approach currently in use by VISTA for high-level conceptual design and concept evaluation. They briefly note how the products of this process feed into the lower-level design and implementation effort. The essence of the design method is an iterative *object decomposition*, which roughly corresponds to functional decomposition in traditional design.

Variations on the object-oriented design theme are being researched in consideration of using OOD with conventional languages, bridging structured analysis and OOD, and streamlining the OOD process. Leading methodology proponents include Booch (1983), Abbott (1983), and EVB Software Engineering (1986).

The object-oriented design approach consists of three primary steps. These are: 1) identify and isolate a *problem statement*, 2) derive an *informal specification* to solve the problem, and, more formally, 3) derive the *object-oriented blueprint*. This approach extends Booch's approach for designing Ada software (1983). We have modified the approach in consideration of the two-segment design process and the review and documentation requirements of the VISTA project. The complete process consists of the following stages:

1. Preparation - Goal Setting

2. Problem Statement

3. Informal Specification

3.1 Specify an informal solution

3.2 Informally review problem and solution

3.3 Refine 2 & 3 until problem and solution are agreed upon

4. Object-Oriented Blueprint

4.1 Identify top-level conceptual objects in the solution space

4.2 Identify object relationships and object dependencies

4.3 Test solution with scenarios

4.4 Descend one level in conceptual design; iterate to 4.1 until level of design has reached appropriate depth (see 1)

5. Detailed Design - Handoff.

The full description of the design approach follows.

4.1 PREPARATION AND GOAL SETTING

Decide on the bounds of the design process to be performed. A particular task or project may require a specific subset of the design process to be performed to verify problem understanding or solution concepts or to sell your solution to a customer. This may mean interrupting design or delaying completion of the design until a later time. If a partial design is the goal, the level of the design to be obtained and the resulting products should be decided upon in advance. Whatever is generated at the end of a partial design must be

able to contribute to completing the design. For both the partial and complete design process, clear and thorough documentation of decisions and design must result. OOD tools discussed in later sections will help in generating design documentation.

4.2 PROBLEM STATEMENT

Generate a short statement of the problem. A single sentence or paragraph should be sufficient to serve as a focal point to the design team summarizing the problem they are to solve. The problem statement is extracted from the user requirements collected as part of the requirements analysis phase of the project. Requirements may have been extracted through structured analysis procedures, "object-oriented analysis," prototyping, or other informal methods. A useful problem definition consists of a 1- to 2-sentence statement of the problem, a 1- to 2- sentence statement of the problem constraints, and a *data dictionary* or an object-based *information model specification* (Shlaer and Mellor 1988) or some other artifact to define the terms encountered in the problem domain and used in the problem statement. The problem description should be read, understood, and agreed upon by all members of the design team.

4.3 INFORMAL SPECIFICATION

Determine what will be done to solve the problem. This phase generates a statement of the solution strategy that is verified by informal review.

4.3.1 Specify an Informal Solution

Generate a description of the solution using a high degree of abstraction. A single or a few paragraphs should suffice for the description. Each statement of the solution should be kept at an equal level of abstraction with each other statement. Interesting or sticky areas of the solution at this point should not result in major digressions for the design team. The statements should describe *what* the solution will contain to address the problem and not

how it will be implemented. Conceptual illustrations and potential user view depictions can be helpful to clarify concepts. The design team should avoid any sort of data or procedural representations at this stage.

An acceptable informal solution description should answer the question: "*If a solution is constructed that follows the strategy described in this paragraph, will the problem be solved?*"

4.3.2 Informally Review Problem and Solution

Verify informal concepts with technical peer groups and users. The review should present the problem, present the solution strategy, and close with some consensus of whether or not the solution strategy, if implemented, will solve the problem. Note that the illustrations and potential user views will be the primary tools in communicating the solution concepts.

4.3.3 Refine the Problem Statement and Informal Specification

Iterate through the preceding steps until problem and solution are agreed upon. At this point, it may be decided that the problem statements need to be reworded or reconsidered. Selection of a more constrained problem or a component part of the original problem is possible at this point without having to sacrifice an overwhelming amount of the design team's time and effort. One important aspect of determining the informal approach is the maintenance of a close association between the design team and the user early in the process so that formative solution concepts are verified. Reviews with technical peer groups will help ensure that the solution concepts are ultimately implementable. Another aspect of the informal specification is that it represents a form of paper prototyping. If the static paper medium is not sufficiently effective in communicating design concepts, animated prototyping tools are available. (*HyperCard*®, *SuperCard*™ and *Prototyper*™ for the Macintosh® are tools that may be used to construct quick concept demonstrations.)

4.4 OBJECT-ORIENTED BLUEPRINT

Perform a top-down, component-wise object decomposition of the solution.

Ultimately, this step will generate a thorough conceptual design specification that is usable to create detailed design specifications. Paper products constituting the specification include: object diagrams or tables, "object-relationship" diagrams, object interface diagrams, user view depictions, and design tests via scenario descriptions.

4.4.1 Identify Top-Level Conceptual Objects in the Solution Space

Identify the key conceptual entities of the system and identify the operations associated with each entity. Remember that, in object-oriented systems, objects are integrated data and procedures. Objects in the solution space are, therefore, entities with the operations they perform. Unfortunately, there are no formal methods for identifying objects and operations in a solution space. The method advanced by Booch involves extracting the nouns and noun phrases from the informal strategy paragraph. Discarding irrelevancies, this would serve as the starting point of an object list. Identification of verbs and verb phrases would provide the initial set of operations on the objects. Attributes of the nouns and verbs may be identified as object characteristics (e.g., a "blue" thing), action characteristics (e.g., a "fast" activity), or a component part of the object. The information collected after parsing the paragraph is consolidated into *object design templates*, more fully described in the tools section below.

4.4.2. Identify Object Relationships and Object Dependencies

Construct object-relationship diagrams to provide overview information about the solution. Construct object-dependency diagrams to show the interfaces and interactions between objects. A fundamental tool for considering the objects in the solution space is an *object-relationship diagram*. The

object-relationship diagram is based on the entity-relationship diagram (Chen 1977) where the objects replacing the entities represent much more powerful system components. This diagram still specifies *what* constitutes the solution space and not *how* the solution functions. An object, specified as part of the solution, plus its component parts plus each of their component parts and so on is an example of a useful structure for an object-relationship diagram.

The informal solution will call out objects that depend upon other objects to accomplish their operations. Also, the designers may discover such relationships, as objects are being designed and recorded, that are not explicitly called out. The relationships between these objects must be captured in a more obvious fashion than in the informal solution description. A *Booch interface diagram* may be used to graphically depict object interfaces plus dependencies between objects. Another tool for capturing objects and dependencies is the CRC (Class, Responsibility, Collaboration) Card. The use of CRC Cards is described in (Beck and Cunningham 1989).

Both the object-relationship diagram and the Booch interface diagram are presented in more detail in the tools section below. Note that a component-based object-relationship diagram, as mentioned above, consists of objects drawn from more than one level of this design process. It may prove more beneficial to the design team to construct one or both of these diagrams at certain points after multiple levels of the object design have been specified. They may then be augmented as the design proceeds.

4.4.3 Test solution with scenarios

Construct scenarios that reflect various aspects of the requirements specification. Test the design by walking through the message generation and responses to high-level function requests. Other tools may be useful at this point to assist in explaining the concepts being presented in the scenario tests. For example, modified state-transition diagrams have been helpful in VISTA scenario tests to illustrate the dynamics of an object that may not have been

clear from the object's description. Potential user view depictions may be helpful at this point. Reviews with the user continue to provide checkpoints so that design concepts conform to specified requirements specification and meet the user's expectations. Note that these user views still represent experimental paper prototyping and do not result in rigorous user interface specifications.

4.4.4 Iterate Until Level of Design Has Reached Appropriate Depth

Continue to identify objects at the next lower conceptual level of the design. Component parts of the top level object were identified as part of the object description. The next step in the design process is to consider these independently, identifying their operations, attributes, and relationships. Continue iteration in this way until the design specification reaches sufficient depth to satisfy the purposes set out at the beginning of the process by the design team. An important question from the "goal setting" step is: *what is a fundamental component of the design?* This must be determined so that you have a specified stopping point. If you have reached a level where your objects are non-decomposable, this is a good stopping point. If you find yourself looking at "objects" that don't do anything--

you are dealing with contributing data elements that are not objects--this also may be a good stopping point. Your team may also reach a point where the team members, the users, and the reviewers all feel comfortable that the design has been adequately captured and that the scenarios work appropriately. This, again, may be time to stop the process and move on the next tasks.

4.5 DETAILED DESIGN - HAND OFF

At this point, the conceptual design can serve as input to the system implementors. However, the implementors need to perform detailed design (now emphasizing *how* the implementation will work rather than *what* the solution accomplishes) to generate the implementation description. A thorough description of this process is beyond the scope of this paper, yet it should be performed in a manner fundamentally different from the conceptual design described here. For object-oriented development, implementors will naturally want to take advantage of the software reusability and existing inheritance structures found in object-oriented development systems. This is a good argument for using the bottom-up approach for detailed design. Meyer (1988) discusses bottom-up strategies for software construction using the object-oriented language, Eiffel.

5.0 TOOLS FOR OBJECT-ORIENTED DESIGN

The VISTA team has created an "object design template" to be used in high-level object definitions. The object template includes the object name, a short paragraph description of the object's purpose within the system (why it is present), a description of the object's characteristics (general object behavior and attributes), a section for a more detailed description of the object's functions or responsibilities (what it does), a section for dependencies or collaborations (who it needs to perform its functions), and a section for unresolved issues (what might be wrong). Figure 3 depicts the object design template.

For the purposes of design, note that objects are defined in two fashions: through an *interface description* or through an *implementation description*. The interface description defines the external interface of an object with information about what messages an object will respond to and what operation the object will respond with. The implementation details of the operation or the object's private data structures are not defined. Conversely, an implementation description provides detailed information about the operation's specification and private data structures used. The two different descriptions serve the two segments of object-oriented design as well as provide programmers with two levels of object description. For the design segments, interface descriptions suffice for conceptual design, whereas implementation descriptions are required for detailed design. For programming, the interface descriptions are sufficient for a programmer wishing to use the service of an object. Implementation details are not needed to be known to him. The implementation description is required by the programmer specifying how a service is to be provided. Internal details may be modified by the implementation programmer; however, because of the interface protection, this will not affect the user of an object's services unless the interface description is changed.

Object relationships may be depicted using an object-relationship diagram. This diagram is a modification of the entity-relationship diagram, popularized by Chen (1977). It is used for graphical presentation of entities and relationships and does not provide a full definition of the entities or the nature of their relationships. For the object-relationship diagram, objects replace the entities and relationships are unchanged. For object-oriented design, this diagram can serve to present objects and their components, and the relationships would denote containment, e.g., Object A requires one or more instances of Object B to perform its tasks. Other relationships can also be modeled with this tool. The same notation used in entity-relationship diagrams for specifying attributes of the relationship (e.g., one-to-one, one-to-many) may still be used. Figure 4 shows a simple example of an object-relationship diagram.

Object dependencies may be diagrammed using a Booch interface diagram. The Booch diagram depicts an abstraction of an object's interface with explicit dependency links between objects drawn in (Figure 5). For objects in the Booch diagrams, the encompassing rectangle represents the boundary of a particular object. Labelled ovals on the left border represent attributes (or *instance variables*) of the object. Labelled rectangles represent the interface description of the object. The variable shape within the rectangle to the right represents hidden procedural details. From this shape emerge the dependency connections between this object and other objects it depends upon to perform its function. The connections are merely lines drawn between objects showing that, for example, Object A depends on Object B to complete its tasks.

VISTA Object Design Template

"AddressBook"

Purpose:

The address book maintains and provides information on colleagues' names, addresses and telephone numbers. We are building this because there is no other tool that satisfies our needs in electronic address books.

Characteristics:

Is maintained alphabetically by entry.

Illustration: **Responsibilities:**

maintains entries
searches entries
prints entries

Collaborations:

entry - workhorse object within the address book.
electronicSecretary - provides new entries

Unresolved Issues:

- When will the electronicSecretary be completed??

Figure 3. Object Design Template.

Figure 4. Object-Relationship Diagram.

Conventional tools are often useful in an object-oriented design. A state-transition diagram, depicting the potential life-cycle of an object, is often helpful in clarifying a conceptual component of the design. The conventional format of the diagram, with nodes representing the states of an object, arcs representing the transitions the object proceeds through, and some specification of the events the object is responding to, is adequate to convey this information. A modification used in the VISTA project is to depict state-transition diagrams in HyperCard® or SuperCard™. States are drawn up on a card, events are programmed into buttons on the card. Mousing on a button implies occurrence of an event, arriving at the resulting card represents a new state of the object. The animation of the state-transition automaton is highly effective in conveying information about an object's life-cycle between designers. It should be remembered that this still represents a conceptual design tool, and the information to be conveyed is conceptual depictions. Still, the HyperCard® or SuperCard™ animation could be

helpful later to implementors who will be looking at this information with software construction in mind.

User views are potential user interface presentations that are a first cut mapping of the designer's model to a user's model. The tools used by designers (such as some of those described above) are not always easily understood by users. In checking that the designer's model of the system still corresponds with the user's requirements, a different presentation medium is needed. User views usually represent a form of paper prototyping of the design model, although HyperCard®, SuperCard™, and other prototyping tools allow quick, high-quality, automated construction of design concepts. The user views should not presume to be well-thought out user interface specifications, rather should be introduced as communication tools. If a user takes a particular liking to a format of the presentation, this can be taken as a fortuitous by-product to be carried into subsequent phases, but should not be a goal of the design team in constructing user views.

Figure 5. Booch Interface Diagram.

6.0 CONCLUSIONS

The object-oriented design methodology has allowed the VISTA project team to successfully perform the conceptual design of large, complex components in a systematic manner. Top-down object decomposition allowed the team to successfully consider design aspects beginning with a global viewpoint of the problem and descending in a controlled fashion to more detailed levels. It has been mentioned in the literature that object-oriented design cannot occur in a top-down fashion because crucial design decisions are made far too early in the design process and can be altered later only at great expense (Meyer 1988). In our experience, using the top-down approach has not jeopardized the implementation, because it

has been acknowledged and planned that the implementation description will be constructed following a different method (e.g., bottom-up, exploratory) to take further advantage of the benefits of the object-oriented paradigm.

Development of the object-oriented design approach has also resulted in the creation of new tools for design (object design templates) and adaptations of conventional tools for use with objects (object-relationship diagrams, HyperCard®- or SuperCard™-based state-transition diagrams). These tools generate a valuable set of documentation capturing the conceptual design and the design decisions made along the way.

6.0 CONCLUSIONS

has been achieved. The design methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system.

Development of the system-oriented design methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system.

The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system. The methodology has been shown to be a useful tool in the design of a system.

7.0 REFERENCES

- Abbott, R. 1983. "Program Design by Informal English Descriptions." *CACM*, 26(11):pp. 882-894.
- Beck, K and Cunningham, W. 1989. "A Laboratory for Teaching Object-Oriented Thinking." *OOPSLA '89 Conference Proceedings*. ACM Press, New York.
- Booch, G. 1983. *Software Engineering with ADA*. Benjamin/Cummings Publishing Company, Menlo Park, California.
- Chen, P. 1977. *The Entity-Relationship Approach to Structured Design*. Q.E.D. Information Sciences, Wellesley, Massachusetts.
- Constantine, L., Myers, G., and Stevens, W. 1974. "Structured Design." *IBM Systems Journal*, 13(2):115-139.
- Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts.
- Dahl, O., Myrhaug, B., and Nygaard, K. 1968. *Simula 67 Common Base Language*. Norwegian Computing Center, Norway.
- EVB Software Engineering. 1986. *Object-Oriented Design Handbook*. Rockville, Maryland.
- Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and its Environment*, Addison-Wesley, Reading, Massachusetts. (Blue book)
- Jackson, M. 1975. *Principles of Program Design*. Academic Press, New York, New York.
- Meyer, B. 1988. *Object-Oriented Software Construction*. Prentice-Hall, New York, New York.
- Shlaer, S. and Mellor, S. 1988. *Object-Oriented Systems Analysis*. Yourdon Press, Englewood Cliffs, New Jersey.
- Stroustrup, B. 1989. "C++ Programming." In *Software Development '89 Proceedings*. Miller Freeman Publications, Burlingame, California.
- Warnier, J. 1974. *Logical Construction of Programs*. Van Nostrand Rheinhold, New York, New York.
- Wiener, R. and Sincovec, R. 1984. *Software Engineering with Modula-2 and Ada*, Wiley, New York, New York.
- Wirth, N. 1971. "Program Development by Stepwise Refinement." *CACM*, (4):pp. 221-227.
- Yourdon, E. and Constantine, L. 1987. *Structured Design*. Yourdon Press, Englewood Cliffs, New Jersey.

APPENDIX A

OBJECT-ORIENTED DESIGN EXAMPLE

APPENDIX A

OBJECT-ORIENTED DESIGN EXAMPLE

The following is an excerpt from an object-oriented design performed at PNL in designing a component of the VISTA system. The context for this example is the design of a tool to assist users in organizing and performing complex procedures that are to be carried out on a computer or with the aid of a computer--essentially, an electronic lab notebook.

Problem Statement

In this example, the design team had come to an important point of the design. The component that a user would be focusing on to complete a subtask needed to be considered. After some discussion, the concept was succinctly captured in the problem statement:

"We need a thing that will give us a focal point for a process to be run -- it should automatically maintain a record of the process runs and results, and allow us to record observations for auditing purposes."

Informal Solution

The informal solution also required much discussion, and its essence is captured here. Note, we still attempt to maintain an even level of abstraction, even though we are in the midst of the object decomposition:

"A topic is the basic entity of the lab notebook. It contains the executable specification of how a process is carried out, maintains an automated history and allows user annotation."

Object Design Template

In decomposing "topic", much of the design information that was discussed in the design sessions needed to be captured outside of the somewhat unstructured meeting minutes. The object design templates served this purpose nicely. As the information was being discussed, a member of the design team captured the information on a white board. It was later cleaned up and entered via object design templates into our computer-based documentation set. By the end of the conceptual design session, we had several pages of design templates for the laboratory notebook.

VISTA Object Design Template

"Topic"

Purpose:

A topic is the basic entity of the lab notebook. It contains the executable specification of how a process is carried out, maintains an automated history and allows user annotation.

Characteristics:

A topic can be viewed similarly to a node in a data flow diagram--the node representing a process and information flowing from children nodes to the Topic to its parent nodes.

Illustration: **Responsibilities:**

provides various views of topic information
executes itself
records information about its execution
allows the user to record information about its execution

Collaborations:

Annotation
Script

Unresolved Issues:

- How/when does the user trigger execution of the topic?

Object-Relationship Diagram

As the object decomposition of "topic" proceeded, it became clear that a diagram capturing an overview of the component relationships was needed. The relationship between topic and other components of the lab notebook was captured in an object-relationship diagram.

Booch Interface Diagram

The other vital set of relationships that needed to be captured was that between an object and those objects it depends upon to complete its tasks. Again, by the end of the conceptual design session, several pages of interface diagrams had been generated.

DISTRIBUTION

No. of

Copies

OFFSITE

2 DOE/Office of Scientific and Technical
Information

ONSITE

2 DOE Richland Operations Office
D. K. Jones
D. E. Trader

42 Pacific Northwest Laboratory

G. D. Black
R. L. Cheatham
G. Chin
C. R. Cole
P. J. Cowley
C. E. Cowan
A. L. Corrigan
V. L. Crow
D. M. DeVaney
B. A. Garrett
C. A. Geffen
L. H. Gerhardstein
M. J. Graham (5)
B. L. Hoopes
E. W. Kleckner
J. R. Lewis
R. B. Melton
M. Olander
J. T. A. Roberts
M. R. Siegal
J. J. Thomas
M. B. Walter
M. A. Whiting (10)
Publishing Coordination
Technical Reports File (5)