546-61
167070
11P.

N89-16325

WA 544

OBJECT-ORIENTED DEVELOPMENT

by

Donald G. Firesmith
Software Methodologist

Magnavox Electronic Systems Company
Advanced Software Systems Division
1313 Production Road
Fort Wayne, IN 46808
(219) 429-4327

1) WHY IS OBJECT-ORIENTED DEVELOPMENT (OOD) IMPORTANT?

OOD is one of the extremely few software development methods actually designed for modern Ada (*) language, real-time, embedded applications.

OOD is a significant improvement over more traditional functional decomposition and modeling methods in that OOD:

a) Better manages the size, complexity, and concurrancy of today's systems.
b) Better addresses important software engineering principles such as abstract data types, levels of abstraction, and information hiding.
c) Produces a better design that more closely matches reality.
d) Produces more maintainable software by better localizing data and thus limiting the impact of requirements changes.
e) Specifically exploits the power of Ada.

2) WHAT IS OOD?

OOD is a systematic, step-by-step software development method that:

a) Has an optimal domain of application -- the development of modern Ada applications (e.g., real-time, embedded software).
b) Covers all, or a major portion, of the software life-cycle.
c) Supports extensive parallel development.
d) Manages the complexity of large development efforts.
e) Is supported by detailed standards and procedures.
f) Requires training and support to be effective.

OOD is:

a) Object-oriented.
b) Ada-oriented.

(*) Ada is a registered trademark of the U.S. Government (AJPO).

c) Based upon modern software engineering.
d) Recursive, globally top-down, hierarchical COMPOSITION method.
e) Revolutionary in approach.
f) A "grab and go" method.
g) Relatively easy to learn.
h) Being successfully used by several companies.
i) Still evolving (see Figure 1).

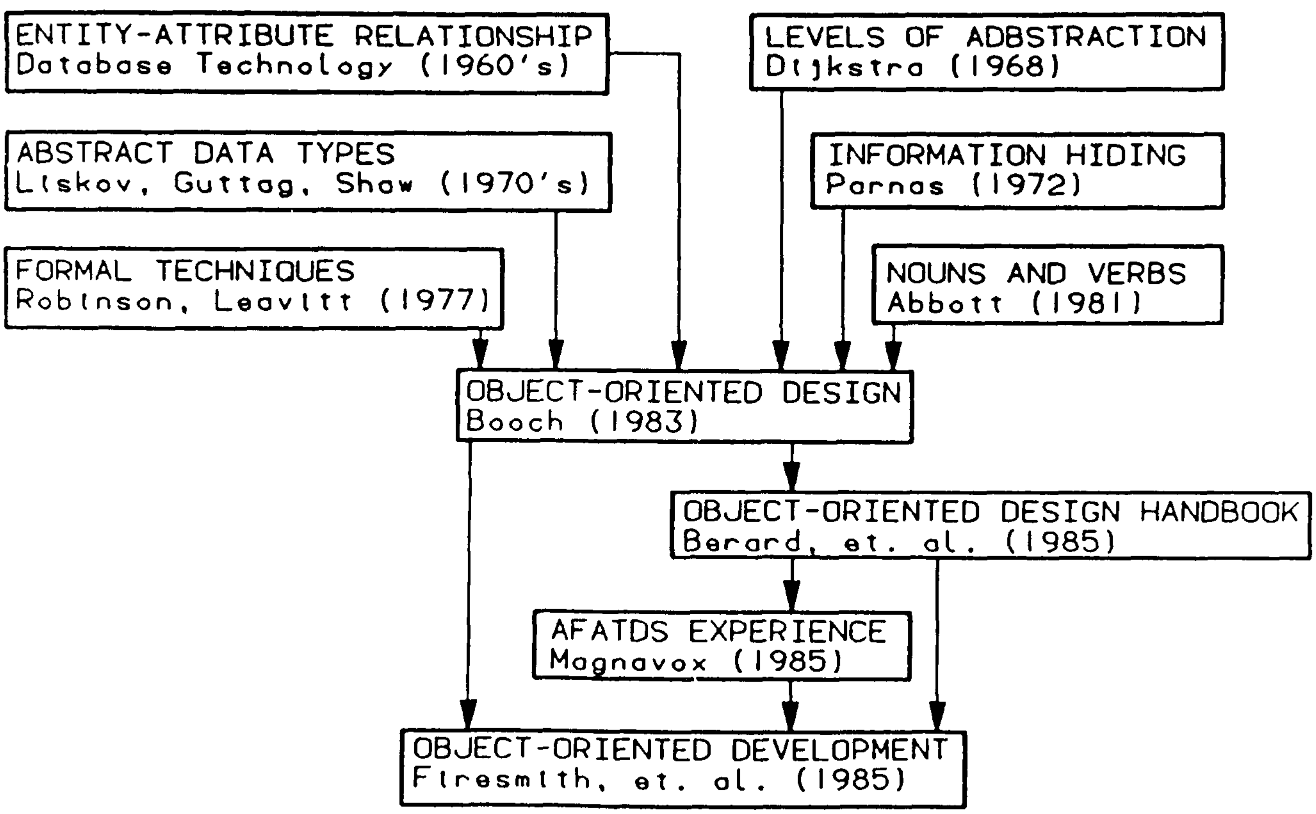

Figure 1: The evolution of OOD

OOD is NOT:

a) A functional, hierarchical DECOMPOSITION method.
b) A modeling method.
c) Easily mated with such methods.
d) Effective without adequate training.
e) Constrained to the classical "waterfall" lifecycle.
f) Consistent with DOD-STD-2167 and related pre-Ada standards.
g) Standardized across the industry.
h) Yet adequately supported by commercially available software tools.

3) OOD IS OBJECT-ORIENTED.

An OBJECT is an entity that:

a) Has a value (e.g., data) or state (e.g., Ada task).
b) Suffers and/or causes operations.

OOD produces:

a) Ada objects that correspond to objects in the real world.
b) Ada types (i.e., object templates).
c) Operations that operate on these objects.

OOD emphasizes the implementation of objects in terms of ABSTRACT DATA TYPES. OOD groups, in the same Ada package:

a) A single type and
b) All operations that operate upon such objects.

OOD produces a substantially different software architecture than traditional functional decomposition methods such as Structured Design which generate units, each of which implements some FUNCTION of the requirements specification.

4) OOD IS ADA-ORIENTED.

Ada is an object-oriented high-level language.

Packages, which are the main building blocks of properly designed Ada software, are also the main building blocks produced by OOD.

The physical design produced by OOD is top-down in terms of Ada:

a) Nesting and
b) Context (i.e., the Ada "with" statement).

OOD separately develops Ada specifications and bodies.

OOD's low-level testing naturally accounts for Ada compilation order constraints.

OOD Diagrams clearly identify the various Ada programming units.

Ada PDL is an integral part of OOD's design and coding steps.

The objects produced by OOD are implemented in Ada as:

a) Constants and variables
b) Abstract data types
c) Tasks

The operations produced by OOD are implemented in Ada primarily as:

a) Subprograms
b) Task entries

5) OOD IS BASED UPON MODERN SOFTWARE ENGINEERING.

OOD specifically addresses each of the following software engineering principles or concepts:

a) ABSTRACT DATA TYPES.
b) ABSTRACTION LEVELS.
c) Cohesion.
d) Concurrency.
e) Coupling.
f) INFORMATION HIDING.
g) Localization.
h) MAINTAINABILITY.
i) MODULARITY.
j) Organizational Independence.
k) Readability.
l) Reusability.
m) Structure.
n) Testability
o) Verifiability.

6) OOD IS RECURSIVE, GLOBALLY TOP-DOWN, HIERARCHIAL COMPOSITION METHOD.

Traditional software development methods are restricted to the classic "waterfall" life-cycle (see Figure 2) in which:

a) The software requirements are analyzed first.
b) The preliminary design is developed second.
c) The detailed design follows.
d) And so on.

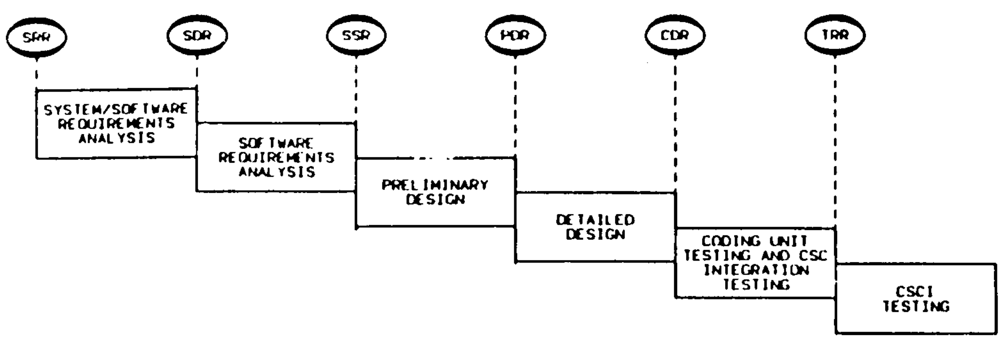

Figure 2: The classic "waterfall" life-cycle

Because the software is developed in a top-down manner only within the boundries of each life-cycle phase, these methods are at best only locally top-down.

OOD is a recursive, globally top-down, hierarchial composition method. Its software life-cycle (see Figure 3) differs significantly from the classic "waterfall" life-cycle because it is based upon recursion and two concepts unique to OOD: the Booch and Subbooch.

A BOOCH is the collection of all software resulting from the recursive application of OOD to a specific set of coherent software requirements -- requirements that specify a single well-defined problem.

A SUBBOOCH is a small, managable subset of a booch that is identified and developed during a single recursion of OOD. See Figure 4.

Note that these two concepts have no obvious natural relationship to the DoD hierarchical decomposition entities CSCI, TLCSC, and LLCSC.

Beginning with the highest abstraction level and progressing steadily downwards in terms of nesting and "withing", the booch is designed, coded, and tested in increments of a subbooch. Thus, the software grows top-down, subbooch by subbooch, via the recursive application of OOD until the entire software tree is completed.

Locally, however, OOD employs the appropriate technique (top-down or bottom-up) depending upon the specific requirements of each individual development activity.

This allows very significant parallel development based upon the "Design a little, code a little, test a little" concept.

7) RESPONSIBILITIES.

The following personnel have OOD responsiblities (see Figure 5):

a) Management
b) Software Development Teams, each consisting of a:
   - Designer
   - Programmer
   - Tester
c) Metrics Collectors
d) Software Quality Evaluation
e) Software System Engineering

8) SUBBOOCH DEVELOPMENT

The subboochs that comprise each booch are recursively developed in a globally top-down fashion. The development of each subbooch consists of the following three subphases:

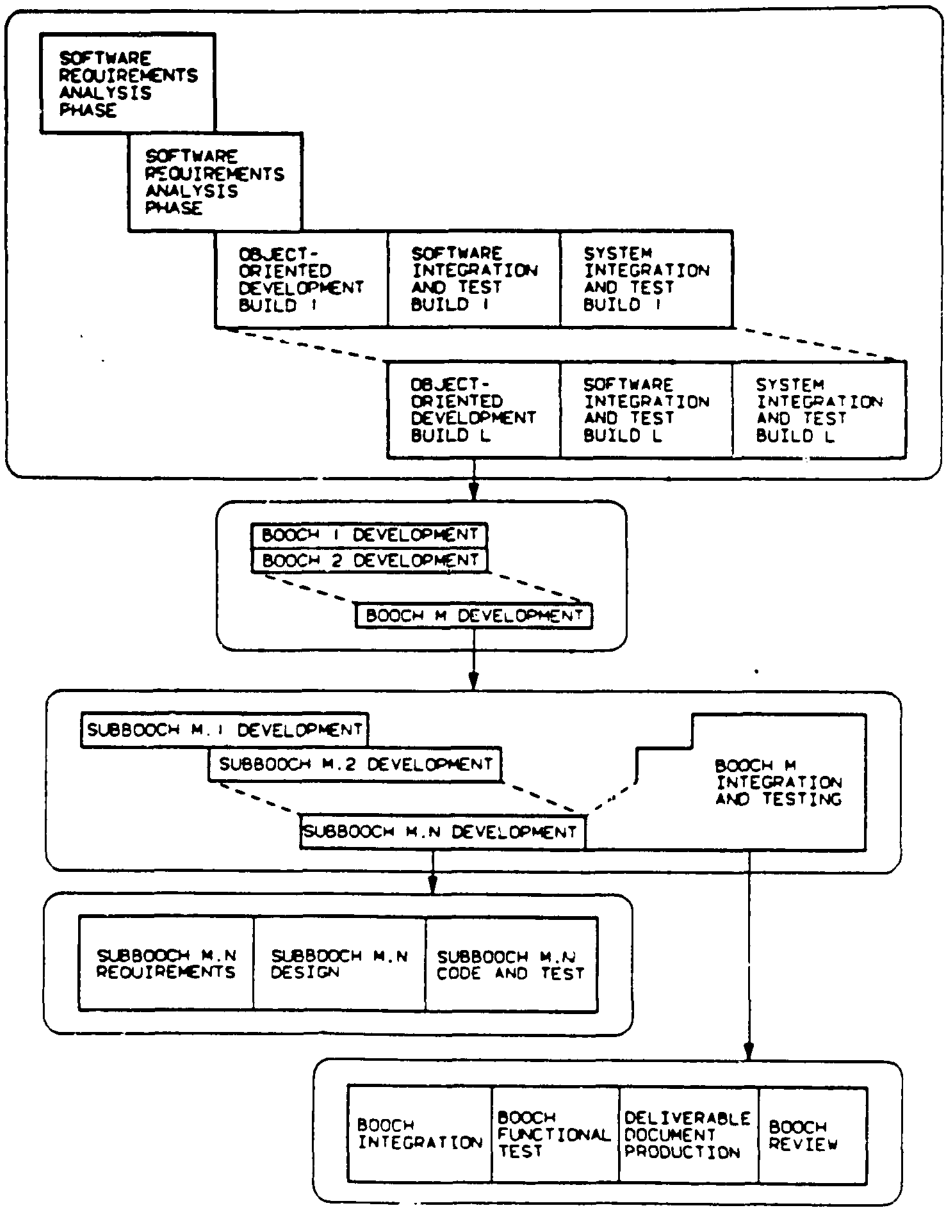

Figure 3: The OOD software life-cycle

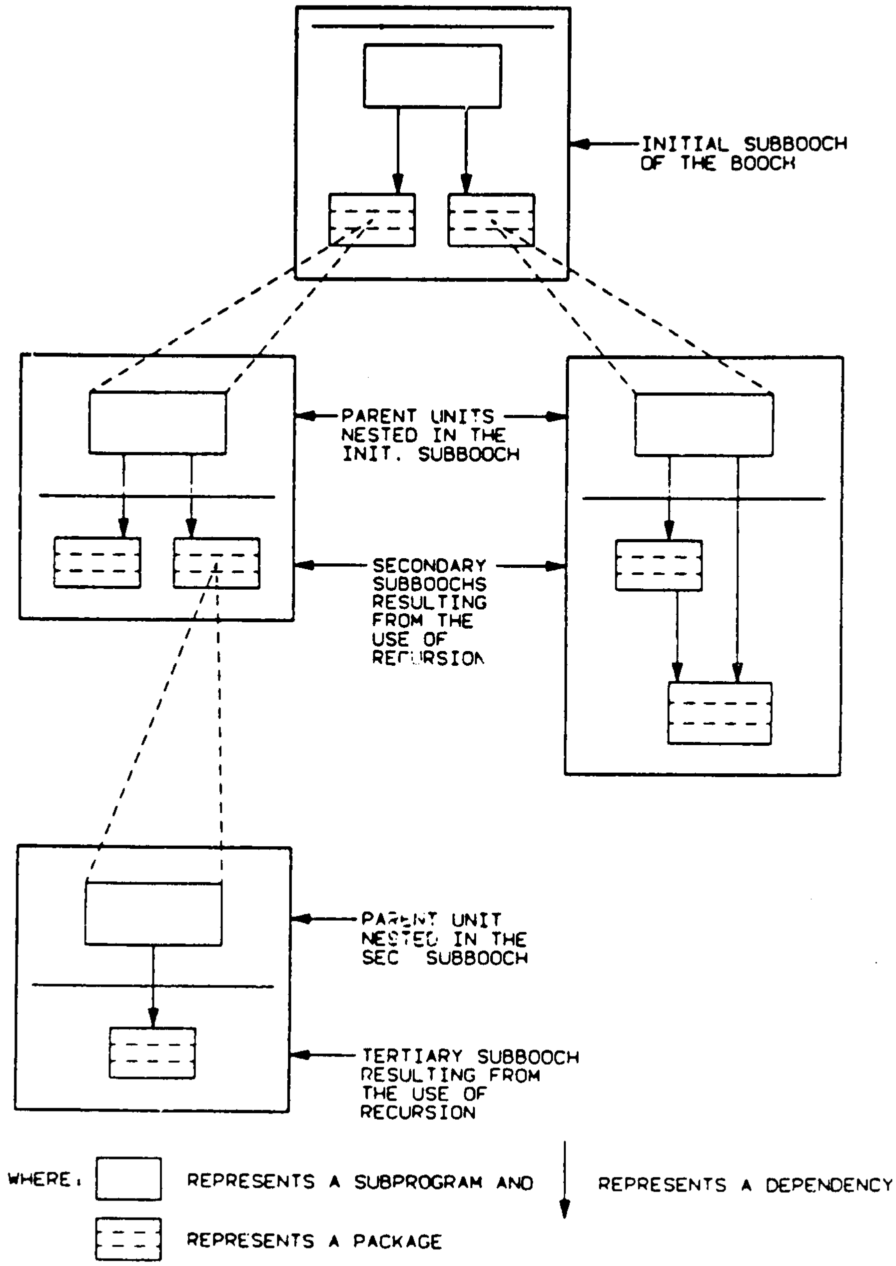

Figure 4: Sample Booch structure

Object-Oriented Development Process

Step	Title	MGMT	Software Dev. Team D	Software Dev. Team P	Software Dev. Team T	MC	SQE
1	INITIATION OF BOOCH DEVELOPMENT	1					4
2	SUBBOOCH DEVELOPMENT						
2.1	SUBBOOCH REQUIREMENTS SUBPHASE						
2.1.1	Initiation of Subbooch Development	1					4
2.1.2	Initiation of the SDF	3	1				4
2.1.3	Problem Statement	3	1	2	2		4
2.1.4	Requirements Analysis	3	1	2	2		4
2.1.5	Subbooch Requirements Inspection	1	2	2	2		4
2.2	SUBBOOCH DESIGN SUBPHASE						
2.2.1	Logical Design	3	1	2	2		4
2.2.2	Object Analysis	3	1	2	2		4
2.2.3	Operation Analysis	3	1	2	2		4
2.2.4	Unit Id., Org., and Dependencies	3	1	2	2		4
2.2.5	Subbooch Preliminary Design Inspection	3	2	1	1		4
2.2.6	Design Analysis	3	1	2	2		4
2.2.7	Coding of Unit Specifications	3	1	2	2		4
2.2.8	Subbooch Detailed Design Inspection	3	2	1	2	1	4
2.3	SUBBOOCH CODE AND TEST SUBPHASE						
2.3.1	Coding of Unit Bodies	3	2	1	2		4
2.3.2	Subbooch Test Plan	3	2	2	1		4
2.3.3	Subbooch Test Software	3	2	2	1		4
2.3.4	Subbooch Test Procedures	3	2	2	1		4
2.3.5	Subbooch Code Inspection	3	1	2	2	1	4
2.3.6	Initial Subbooch Testing	3	2	2	1		4
3	BOOCH INTEGRATION AND TESTING						
3.1	BOOCH INTEGRATION	3			1		4
3.2	BOOCH FUNCTIONAL TESTING	3			1		4
3.3	BOOCH DELIVERABLE DOCUMENTATION	2	1	1	1		4
3.4	BOOCH REVIEW	1	2	2	2	1	1

MGMT = Management
D = Designer(s)
P = Programmer(s)
T = Tester(s)
MC = Metrics Collector(s)
SQE = Software Quality Evaluation

1 = Primary or major responsibility
2 = Secondary responsibility
3 = Managerial responsibility
4 = Independent audit responsibility

FIGURE 5: OOD Responsibilities

a) Subbooch Requirements.
b) Subbooch Design.
c) Subbooch Code and Test.

The SUBBOOCH REQUIREMENTS SUBPHASE has the following steps:

INITIATION OF SUBBOOCH DEVELOPMENT - The Manager initiates subbooch development by identifying the members of the associated Software Development Team and tasking them to meet an assigned schedule of subbooch milestones.

INITIATION OF SOFTWARE DEVELOPMENT FILE (SDF) - The Designer initiates the associated SDF by obtaining an empty SDF binder and inserting the initial Software Engineering Forms (SEFS) that make up the coverpages.

PROBLEM STATEMENT - The Software Development Team jointly state in a single sentence the problem to be solved during the current recursion.

REQUIREMENTS ANALYSIS - The Software Development Team jointly collect, analyze, clarify, organize, and identify the subbooch requirements.

SUBBOOCH REQUIREMENTS INSPECTION - The Designer prepares the SDF for inspection. The Manager schedules the associated meeting. The Manager, the Programmer, and the Tester perform the inspection. The Software Development Team takes any appropriate corrective action.

The SUBBOOCH DESIGN SUBPHASE has the following steps:

LOGICAL DESIGN - The Software Development Team (under the leadership of the Designer) develops in a single paragraph a logical design that properly solves the problem of the current recursion and identifies the relevant objects and operations.

OBJECT ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes all relevant objects in the logical design paragraph, determines and documents their relevancy, and provides the relevant objects with valid Ada identifiers, brief descriptions, and a list of associated attributes.

OPERATION ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes all relevant operations in the logical design paragraph, determines and documents their relevancy, and provides the relevant operations with valid Ada identifiers, brief descriptions, and a list of associated attributes.

MODULE IDENTIFICATION, ORGANIZATION, AND DEPENDENCIES - The Software Development Team (under the leadership of the Designer) organizes all relevant objects and operations by types, identifies the non-nested units for each such organization, nests the organized objects and operations within these units, and determines the visible dependencies between these units.

SUBBOOCH PRELIMINARY DESIGN INSPECTION - The Designer prepares the SDF for inspection. The Programmer and Tester perform the Inspection. The Software Development Team takes any appropriate corrective action.

DESIGN ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes the design, identifies the type of the nested units, common software, and nested units requiring recursion, etc.

CODING OF UNIT SPECIFICATIONS - The Software Development Team (under the leadership of the Designer) implements and compiles, in a bottom-up manner in terms of unit dependencies, the Ada specifications of all units. This includes the development of specification headers, PDL, comments, and code from skeleton unit specifications.

SUBBOOCH DETAILED DESIGN INSPECTION - The Designer prepares the SDF for inspection. The Metrics Collector collects, summarizes, and reports the subbooch design metrics. The Programmer and Tester perform the Inspection. The Software Development Team takes any appropriate corrective action.

The SUBBOOCH CODE AND TEST SUBPHASE has the following steps:

CODING OF UNIT BODIES - The Software Development Team (under the leadership of the Programmer) implements and compiles, in a top-down manner in terms of unit dependencies, the Ada bodies of all units to be implemented during the current build. This includes the development of body headers, PDL, comments, and code from skeleton unit bodies using the technique of step-wise refinement.

SUBBOOCH TEST PLAN - The Software Development Team (under the leadership of the Tester) develops the test plan by determining, creating files of, and documenting the test input and expected test output data required for all subbooch testing and documenting the allocation of these test cases to specific subbooch tests.

SUBBOOCH TEST SOFTWARE - The Software Development Team (under the leadership of the Tester) designs, implements, and compiles all test software programs required for subbooch testing scheduled for the current build.

SUBBOOCH TEST PROCEDURES - The Software Development Team (under the leadership of the Tester) develops the detailed step-by-step procedures for performing all subbooch tests scheduled for the current build.

SUBBOOCH CODE INSPECTION - The Programmer prepares the SDF for inspection. The Metrics Collector collects, summarizes, and reports the subbooch code metrics. The Software Development Team perform the inspection. The Software Development Team takes any appropriate corrective action.

INITIAL SUBBOOCH TESTING - The Software Development Team (under the leadership of the Tester) perform and document the results of all initial subbooch tests.

9) PRACTICAL EXPERIENCE.

The use of OOD at Magnavox on the AFATDS Project (over 50K lines of Ada code so far) has resulted in the following lessons learned:

a) Avoid overspecifying the requirements with explicit or implicit design information of a functional decomposition nature.

b) If a functional decomposition method is used to produce the top-level design, it will be incompatible with the design produced by OOD at the lower-levels and numerous interface problems will result.

c) Replacing the previous functional decomposition mindset is difficult, primarily among the more experienced designers.

d) The concept of recursion is fairly difficult to master.

e) OOD training and support in the method needs to continue beyond the classroom.

f) OOD needs to be further refined, primarily in the area of object-oriented requirements analysis.

g) Ada-oriented test training is as necessary as training in Ada-oriented design and programming.

h) OOD improves designs due to:

- Proper abstraction levels.
- Proper information hiding.
- High modularity.
- Improved interfaces.
- Good support for strong typing.
- Good correspondance to the real world.

i) OOD improves productivity due to:

- Enhanced parallel development.
- Reuse of code.
- Easy coding from design information.
- Easy modification of design and code.