N89-16330

TOWARDS A GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT METHODOLOGY

Ed Seidewitz / Code 554
Mike Stark / Code 552
Goddard Space Flight Center
Greenbelt, MD 20771

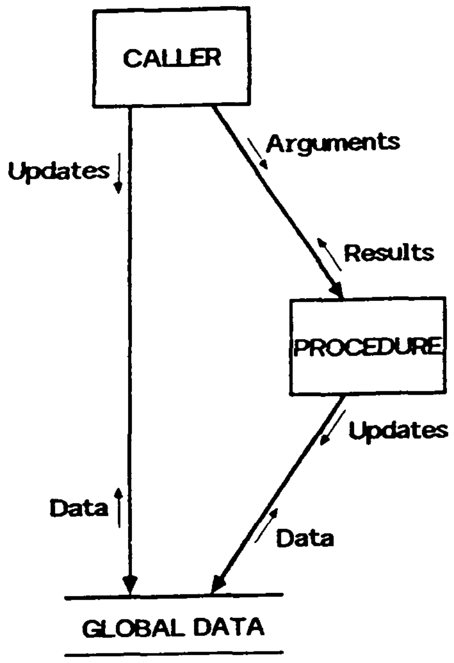
1. INTRODUCTION

An object is an abstract software model of a problem domain entity. Objects are packages of both data and operations on that data [Goldberg 83, Booch 83]. The Ada (tm) package construct is representative of this general notion of an object. Object-oriented design is the technique of using objects as the basic unit of modularity in system design. The Software Engineering Laboratory at the Goddard Space Flight Center is currently involved in a pilot project to develop a flight dynamics simulator in Ada (approximately 40,000 statements) using object-oriented methods. Several authors have applied object-oriented concepts to Ada (e.g., [Booch 83, Cherry 85b]). In our experience we have found these methodologies limited [Nelson 86]. As a result we have synthesized a more general approach which allows a designer to apply powerful, object-oriented principles to a wide range of applications and at all stages of design. The present paper provides an overview of our approach. Further, we also consider how object-oriented design fits into the overall software life-cycle.

2. OBJECTS AND OBJECT DIAGRAMS

We can model a procedure as a mathematical function. That is, given a certain set of inputs (arguments and global data), a procedure always produces the same set of outputs (results and global updates). A procedure, for example, cannot directly model an address book, because an address book has memory (a set of addresses) which can be accessed and updated. Normally, the solution to this is to place such memory in global variables.

Figure 1 gives a representation of the above situation. This diagram uses a notation similar to [Yourdon 79] to show both data and control flow. The arrow from CALLER to PROCEDURE indicates that CALLER transfers control to PROCEDURE. Note that there is an implicit return of control when PROCEDURE finishes. The smaller arrows in figure 1 show the data flows, which may go in either direction along the control arrow. Also, figure 1 includes an explicit symbol for the GLOBAL DATA. Control arrows directed towards this symbol denote data access, even though control never really flows into the data, of course. This convention indicates that the data is always passive and never initiates any action.



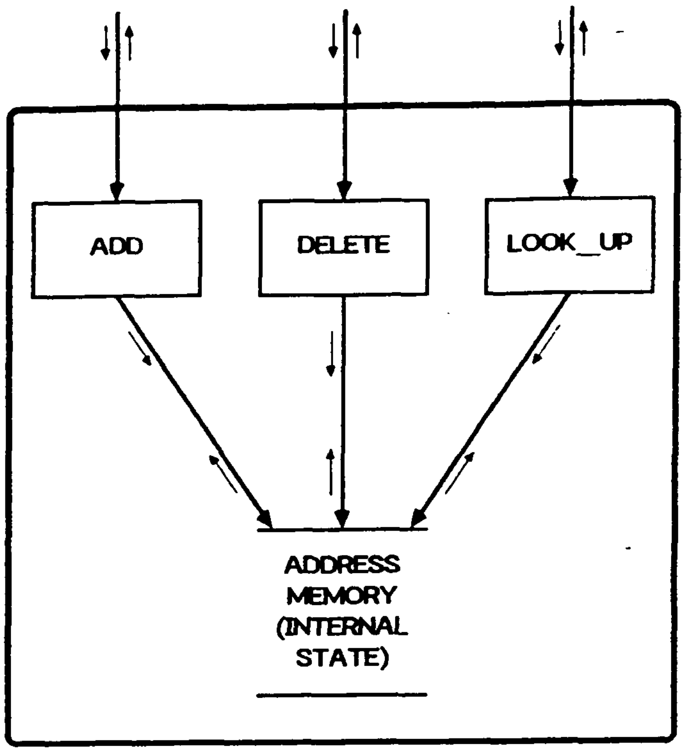


FIGURE 1 A procedure call

FIGURE 2 An ADDRESS BOOK object

The use of global storage leaves data open to illicit modification. To avoid this, an object packages some memory together with all allowable operations on it. We can model an object as a mathematical "state machine" with some internal state which can be accessed and modified by a limited number of mathematical functions. We thus implement an object as a packaged set of procedures and internal data, as shown in figure 2. For an address book object, the internal memory would be a set of addresses, and the allowable operations would be accessing an address by name, adding an address, etc. Unlike a procedure, the _same_ arguments to an object operation may produce _different_ results at different times, depending on the hidden internal state. We will diagram an object showing only its operational connections to other objects, as in the _object diagram_ of figure 3 [Seidewitz 85a].

When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows or all individual procedure control flows. Therefore, an arrow between objects on an object diagram indicates that one object invokes _one or more_ of the operations provided by another object and is _not marked_ with data flow arrows. _Object descriptions_ for each object on a diagram provide details of the data flow. An object description includes a list of all operations provided by an object and, for each arrow leaving the object, a list of operations used from another object. For example, the object description for DATE BOOK from figure 3 is:

```
    Provides:
      Next_Appointment () NAME + ADDRESS
      Get_Appointment (DATE + TIME) NAME + ADDRESS
      Make_Appointment (DATE + TIME + NAME)
      Cancel_Appointment (DATE + TIME)

    Uses:

      ADDRESS BOOK
        Look_Up

      CLOCK
        Get_Date
        Get_Time
```

Data in parentheses are arguments which flow along the control arrow, while unparenthesized data are results which are returned.

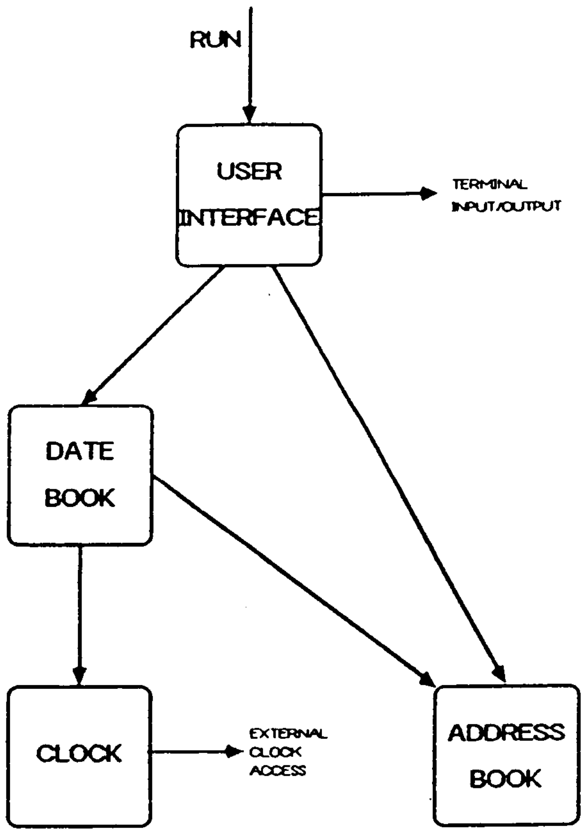


FIGURE 3 A simple schedule organizer

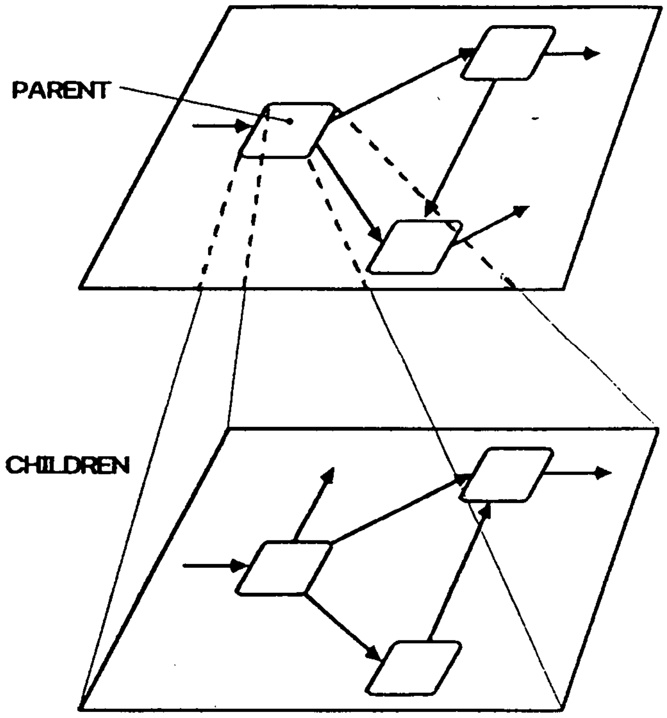


FIGURE 4 Parent-child hierarchy

3. OBJECT-ORIENTED DESIGN

The intent of an object is to represent a problem domain entity. The concept of abstraction deals with how an object presents this representation to other objects [Dijkstra 68, Liskov 74, Booch 83]. There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence. The following are some points in this scale:

Best

Entity Abstraction - An object represents a useful model of a problem domain entity.

Action Abstraction - An object provides a generalized set of operations which all perform the same kind of function.

Virtual Machine Abstraction - An object groups together operations which are all used by some superior level of control or all use some junior level set of operations.

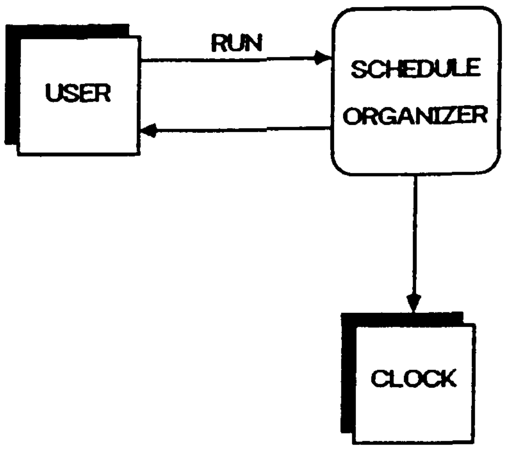
Worst

Coincidental "Abstraction" - An object packages a set of operations which have no relation to each other.

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of information hiding states that such details should be kept secret from other objects [Parnas 72, Booch 83], so as to better preserve the abstraction modeled by the object.

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design [Seidewitz 85b]. Following [Rajlich 85], we consider two orthogonal hierarchies in software system designs. The parent-child hierarchy deals with the decomposition of larger objects into smaller component objects. The seniority hierarchy deals with the organization of a set of objects into "layers". Each layer defines a virtual machine which provides services to senior layers [Dijkstra 68]. A major strength of object diagrams is that they can distinctly represent these hierarchies.

The parent-child hierarchy is directly expressed by leveling object diagrams (see figure 4). At its top level, any complete system may be represented by a single object. For example, figure 5 shows a diagram of the complete SCHEDULE ORGANIZER of the last section. The object SCHEDULE ORGANIZER represents the "parent" of the complete object diagram of figure 3. The boxes labeled "USER" and "CLOCK" are external entities, objects which are not included in the system, but which communicates with the top level system object. Note the arrow labeled "RUN". By convention, RUN is the operation used to initially invoke the entire system.



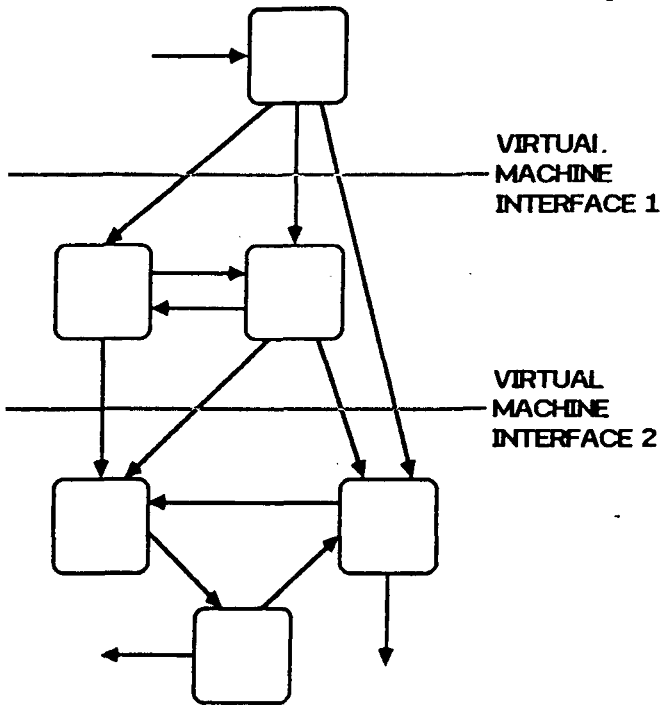


FIGURE 5 External Entities Diagram

FIGURE 6 Seniority hierarchy

Figure 3 is the decomposition of the SCHEDULE ORGANIZER of figure 5. Beginning at the system level, each object can be refined in this way into a lower level object diagram. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into primitive objects, procedures and internal state data stores, resulting in diagrams similar to figure 2.

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 6). Any layer in a seniority hierarchy can call on any operation in junior layers, but never any operation in a senior layer. Thus, all cyclic relationships between objects must be contained within a virtual machine layer. Object diagrams are drawn with the seniority hierarchy shown vertically. Each senior object can be designed as if the operations provided by junior layers were "primitive operations" in an extended language. Each virtual machine layer will generally contain several objects, each designed according to the principles of abstraction and information hiding.

The main advantage of a seniority hierarchy is that it reduces the coupling between objects. This is because all objects in one virtual machine layer need to know nothing about senior layers. Further, the centralization of the procedural and data flow control in senior objects can make a system easier to understand and modify. However, this very centralization can cause a messy bottleneck. In such cases, the complexity of senior levels can be traded off against the coupling of junior levels. The important point is that the strength of the seniority hierarchy in a design can be chosen from a spectrum of possibilities, with the best design generally lying between the extremes. This gives the designer great power and flexibility in adapting system designs to specific applications.

In the simple automated plant simulation system shown in figure 7, the junior level components do not interact directly. This design is somewhat like an object-oriented version of the structured designs of [Yourdon 79]. We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual machine layer (see figure 8). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow. We can even remove the senior object completely by distributing control among the junior level objects (see figure 9). The splitting of the RUN control arrow in figure 11 means that the three objects are activated simultaneously and that they run concurrently. The seniority hierarchy has collapsed, leaving a homologous or non-hierarchical design [Yourdon 79] (no seniority hierarchy, that is; the parent-child hierarchy still remains). A design which is homologous at all decomposition levels is very similar to what would be produced by the PAMELA (tm) methodology of [Cherry 85a, Cherry 85b].

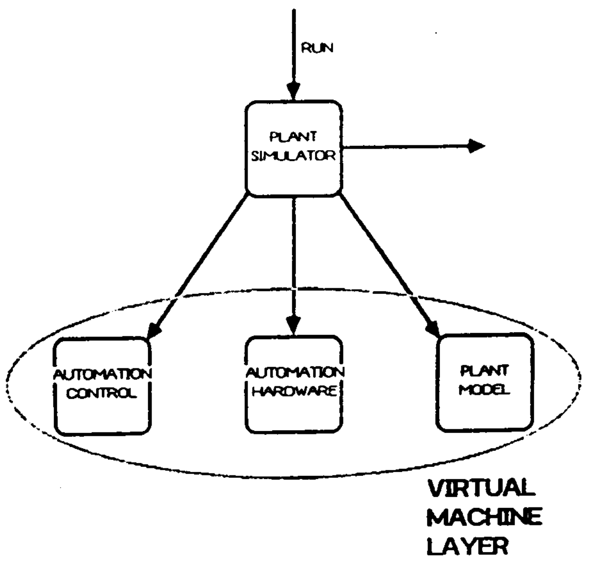


FIGURE 7 A simple plant automation simulation system

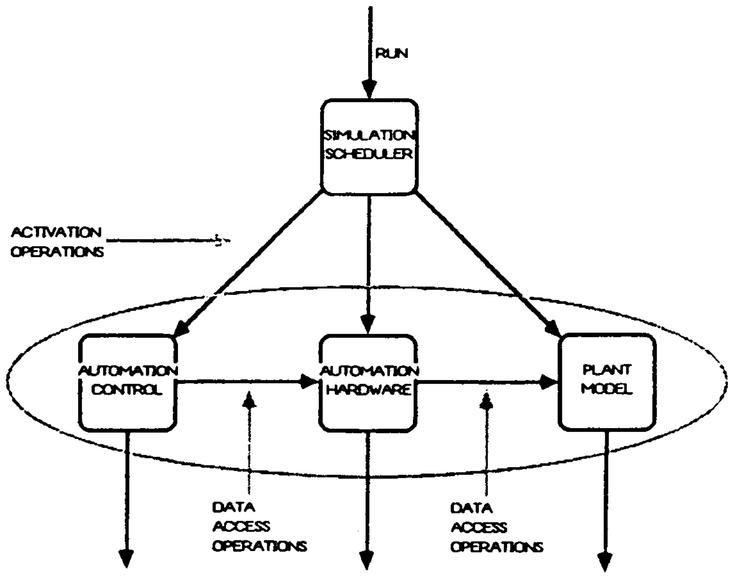
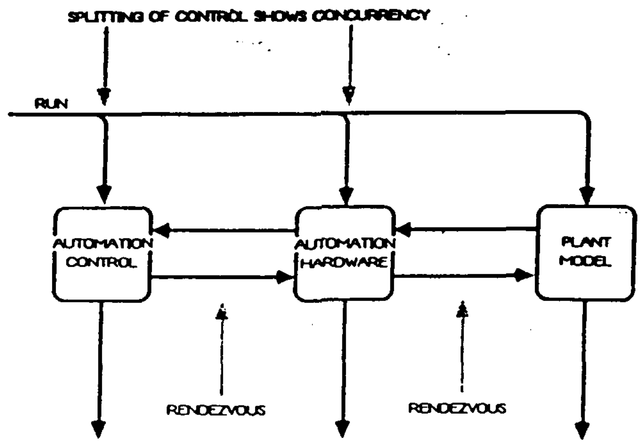


FIGURE 8 plant simulator with junior-level connections



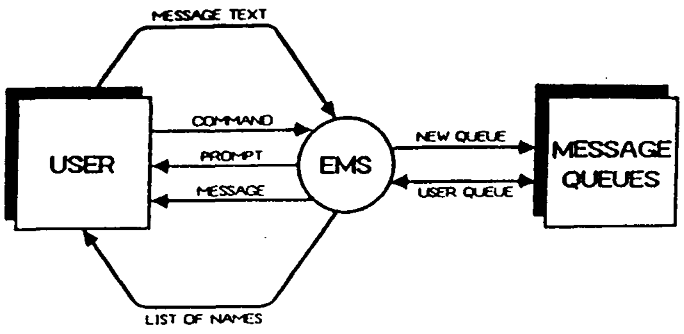


FIGURE 10 EMS context diagram

FIGURE 9 Plant simulator, homologous design

4. OBJECT-ORIENTED LIFE CYCLE

Object diagrams and the object-oriented design concepts discussed above can be used as part of an object-oriented life cycle. To do this, we must show that a specification can be translated into object diagrams, and that object diagrams map readily into Ada. We use structured analysis for developing the specification [DeMarco 79]. The data flow diagrams of a structured specification provide a leveled, graphical notation containing the information needed to represent abstract entities, but in a form emphasizing data flow and data transformation.

Abstraction analysis is the process of making a transition from a structured specification to an object-oriented design [Stark 86]. We will use a simplified version of an Electronic Message System (EMS) as an example of abstraction analysis. Figure 10 is the context diagram for EMS, and Figure 11 is the level 0 data flow diagram. EMS must allow the user to send, read, and respond to messages, to obtain a directory of valid users to which messages can be sent, and to add and delete users from that directory.

The first step of abstraction analysis is to find a central entity. This is the entity that represents the best abstraction for what the system does or models. The central entity is identified in a similar way to transform analysis [Yourdon 79], but instead of searching for where incoming and outgoing data flows are most abstract we look for a set of processes and data stores that are most abstract. It may sometimes be necessary to look at lower level data flow diagrams to find the central entity. EMS is a system serving a person sitting at a terminal sending and receiving messages. On figure 11 we have circled the "current user" data store and the process 1.0 GET EMS COMMAND. Together this process and data store represent the user entering commands at a terminal. Thus they represent the central entity.

Next, we need to find entities that directly support the central entity. We do this by following data flows away from the central entity and grouping processes and data stores into abstract entities. In our example the USER DIRECTORY data store and the three processes (2.0, 4.0 and 5.0) supporting it form an entity. The process 3.0 ACCESS QUEUES with the data store USER QUEUE INDEX also form an entity. All these entities are circled and labeled on figure 11. We continue to follow the data flows and to identify entities until all the processes and data stores are associated with an entity.

Figure 12 is the entity graph for EMS. Squares represent entities, lines with arrows represent flow of control from one entity to another, and lines with no arrowhead represent interactions where flow of control is not yet determined. A "most senior" entity is placed into the design to give an initial flow of control. In the EMS example, entity EMS is this most senior object, and we have the USER INTERFACE entity "controlling" the external entity USER. This flow of control

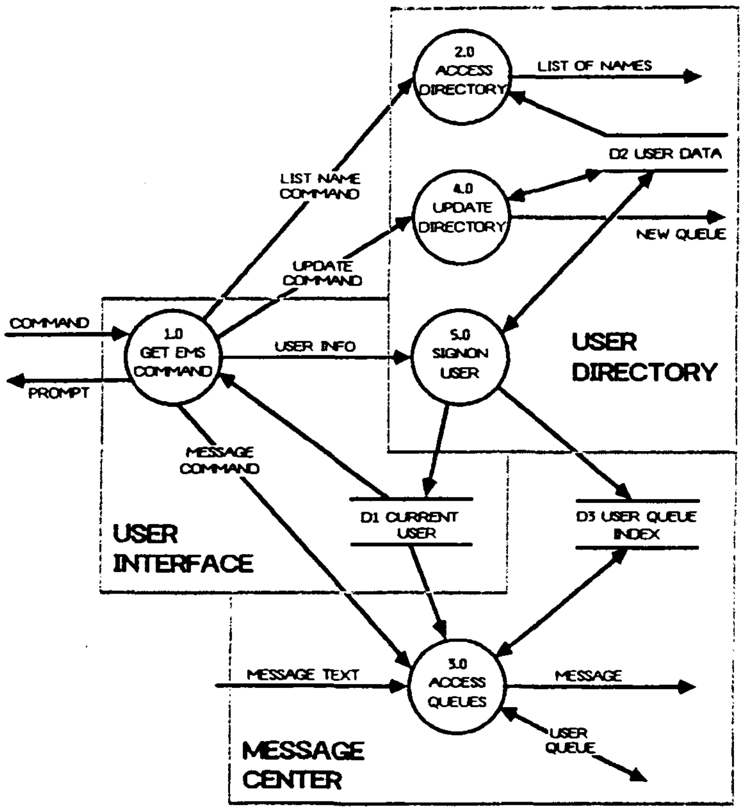


FIGURE 11 EMS level 0 data flow diagram

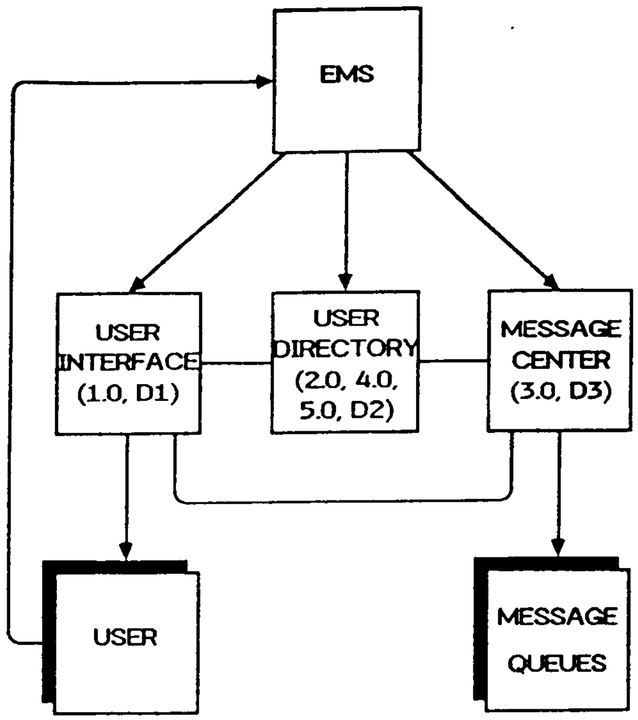


FIGURE 12 EMS entity graph

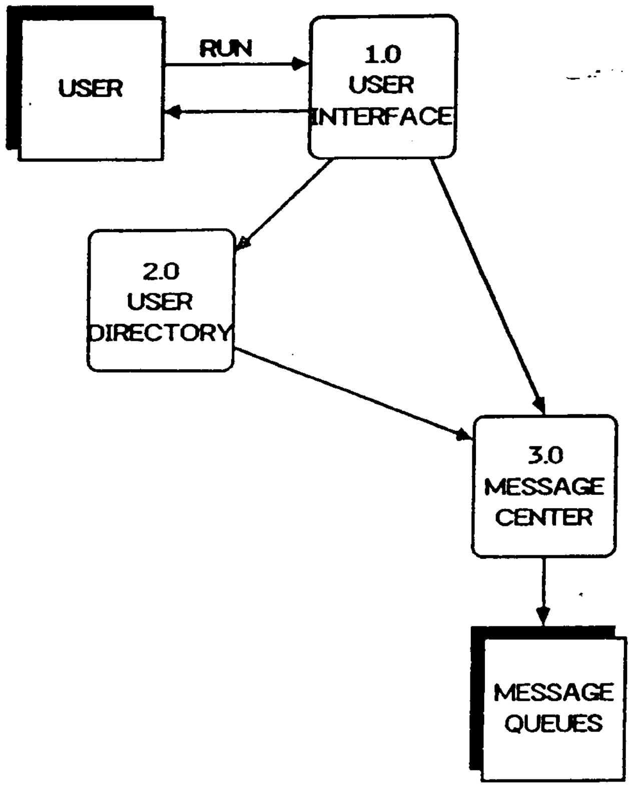


FIGURE 13 EMS object diagram

into USER will ultimately be implemented as read and write operations. Note also that the USER entity controls EMS. This flow of control represents the user invoking the EMS system. After this invocation control resides with EMS until the system is exited. All other potential interfaces are shown by lines with no arrows. The numbers inside the squares represent the processes and the data stores contained in the entity. This provides traceability from requirements to design.

The entity graph is the starting point for object identification. It shows entities with the highest abstraction possible and also shows all the possible interconnections between the entities. Since we are trying to balance design complexity, object abstraction, and control hierarchy, we will alter the entity graph to form the final object diagram. In EMS the entities are easily mapped into objects. The entities USER, USER INTERFACE, and EMS form a cyclic graph and therefore are on the same virtual machine level. We cannot combine an external entity into an object, but combining EMS and USER INTERFACE yields a single object that is senior to USER DIRECTORY and MESSAGE CENTER. Combining the two junior objects would simplify the design, but at the expense of abstraction, as the message passing mechanisms have little to do with the directory. We have also chosen to make USER DIRECTORY senior to MESSAGE CENTER, since the data flows are from USER DIRECTORY into data stores contained by MESSAGE CENTER. Figure 13 shows the resulting object diagram.

Needless to say, identifying objects is not always this simple. Usually there is a trade-off made between level of abstraction and design complexity, or a balancing of these two considerations and the virtual machine hierarchy. When these situations occur it is still the designer's judgement that must determine which side of the trade-off matters more for the application being designed.

Once the object diagrams are drawn we can identify the operations provided and used by each object. In the case of 2.0 USER DIRECTORY the operations are identified by examining the primitive processes contained within processes 2.0, 4.0 and 5.0 on figure 11. The data exchanged are identified by looking at data flows crossing the object boundaries, with the detailed information about the data being found in the data dictionary. The object description is produced by matching the operations and the data. The description generated for 2.0 USER DIRECTORY is as follows:

```
Provides:
  List_Names () LIST_OF_NAMES
  Add_User (USER_NAME + PASSWORD)
  Delete_User (USER_NAME)
  Signon (USER_NAME + PASSWORD) VALIDITY_FLAG

Uses:
  3.0 MESSAGE QUEUES
      Reset_Queue
      Create_New_Queue
```

Using the subset data flow diagram of processes and data stores that an object contains, the process of object identification can be repeated to produce a child object diagram. The only difference is that entities are identified based on how they support the object's operations, not by finding a central entity. This process is used until the lowest level of data flow diagrams is exhausted.

The transition from an object diagram to Ada is straightforward. The relationship between object diagram notations and Ada language features is:

Object Diagram	Ada Construct
Object	Package
Procedure	Subprogram
State	Package or task variables
Arrow	Procedure/function/entry call
Actor	Entries/Accepts (not covered in this paper)

Package specifications are derived from the list of operations provided by an object. For the EMS USER DIRECTORY object the package specification is:

```
package User_Directory is

  subtype USER_NAME is STRING(1..20);
  subtype PASSWORD is STRING(1..6);
  type LIST_OF_NAMES is array (POSITIVE range <>) of USER_NAME;

  procedure Signon (User: in USER_NAME; PW : in PASSWORD;
    Valid_User : out Boolean);
  procedure Add_User (U: in USER_NAME; PW : in PASSWORD);
  procedure Delete_User (U: in USER_NAME);
  function List_Names return LIST_OF_NAMES;

end User_Directory;
```

The package specifications derived from the level 0 object diagram are placed in the declarative part of the top level Ada procedure as follows:

```
procedure EMS is
  package User_Interface is
    procedure Start;
     ...
  end User_Interface;

  package User_Directory is
     ...
  end User_Directory;

  package Message_Queues is
     ...
  end Message_Queues;

  package body User_Interface is separate;
  package body User_Directory is separate;
  package body Message_Queues is separate;

begin
  User_Interface.Start;
end EMS;
```

For lower level object diagrams the mapping is similar, with package specifications being nested in the package body of the parent object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. To implement the same object diagram with library units would require the addition of a package to contain data types used by two or more objects. This added package would serve as a global data dictionary.

The process of transforming object diagrams to Ada is followed down all the child object diagrams until we are at the level of implementing individual subprograms. If the mapping is done without explicitly creating library units the lowest level subprograms will all be implemented as subunits, rather than by embedding the code in package bodies.

5. EVALUATION OF THE METHODOLOGY

To measure how well abstraction analysis works as a methodology we must first define our criteria for a good methodology. We will use Barry Boehm's "Seven Principles of Software Engineering" [Boehm 76] as a basis of comparison. These principles are:

Manage using a sequential life cycle plan
Maintain disciplined product control
Perform continuous validation
Use enhanced top down structured design
Maintain clear accountability for results
Use better and fewer people
Maintain a commitment to improve the process

Abstraction analysis supports all these principles. The life cycle plan is supported by providing the abstraction analysis method for producing object diagrams, which are in turn mappable into Ada. This also provides a means of disciplined product control by tracing how Ada software implements an object oriented design, and also tracing how the design meets the specification. This traceability allows a manager to see that software meets its specification, and allows maintenance of specifications, design, and software to be consistent. Grady Booch's [Booch 83] work influenced our methodology, but did not provide a sufficient means of specifying large systems. Another drawback is that Booch does not define a formal mapping from a specification to a design.

The graphic notation supports a top down approach to software development. The leveling of both dataflow diagrams and of object diagrams allows the designer to start at a high level and work top-down to a design solution. The use of graphics also supports continuous validation by making design walkthroughs and iterative changes easier tasks to perform. Both Booch and Cherry [Cherry 85b] use graphics, but Booch's notation was not designed for large applications, and Cherry's methodology stops graphing after all the concurrent objects have been identified. The graphics used by structured analysis [DeMarco 79] provide the best analogy to how graphics are used in the object diagram notation.

The life cycle model we have defined also supports the remaining three principles. Objects are defined in the design phase and implemented as separate Ada compilation units. Tools such as unit development folders can be used to maintain accountability for completion of the design, implementation, and testing of objects. It is hoped that the object-oriented approach and the use of Ada will enhance both productivity and software reliability. This assertion will be tested by measuring the outcome of the pilot project in the Software Engineering Laboratory at Goddard Space Flight Center. The success of this methodology would allow better and fewer people to concentrate more effort on producing a good design.

Finally, we are certainly committed to improving the process. The object diagram notation and abstraction analysis have already seen much change since the initial versions were defined. Further refinement will be to define criteria for using parallelism, criteria for choosing between library units and the nested approach defined above, and to generate object-oriented approaches to software specifications and software testing.

6. CONCLUSION

Object diagrams have been used to design a 5000 statement team trainging exercise and to design the entire dynamics simulator. They are also being used to design another 50,000 statement Ada system and a personnal computer based system that will be written in Modula II. Our design methodology evolved out of these experiences as well as the limitations of other methods we studied. Object diagrams, abstraction analysis and associated principles provide a unified framework which encompasses concepts from [Yourdon 79], [Booch 83] and [Cherry 85b]. This general object-oriented approach handles high level system design, possibly with concurrency, through object-oriented decomposition down to a completely functional level. We are currently studying how object-oriented concepts can be used in other phases of the software life-cycle, such as specification and testing. When complete, this synthesis should produce a truly general object-oriented development methodology.

TRADEMARKS

Ada is a trademark of the US Government (Ada Joint Program Office).

PAMELA is a trademark of George W. Cherry.

REFERENCES

[Boehm 76]	Boehm, Barry W. "Seven Basic Principles of Software Engineering," NASA/GSFC Engineering Colloquium, 1976.
[Booch 83]	Grady Booch. Software Engineering with Ada, Benjamin/Cummings, 1983.
[Cherry 85a]	George W. Cherry. PAMELA: Process Abstraction Method for Embedded Large Applications, Course notes, Thought**Tools, January 1985.
[Cherry 85b]	George W. Cherry and Grad S. Crawford. The PAMELA (tm) Methodology, November 1985.
[DeMarco 79]	Tom DeMarco. Structured Analysis and System Specification, Prentice-Hall, 1979.

[Dijkstra 68] Edsgar W. Dijkstra. "The Structure of the 'THE' Multiprogramming System," _Communications of the ACM_, May 1968.

[Goldberg 83] Adele Goldberg and David Robison. _Smalltalk 80: The Language and Its Implementation_. Addison-Wesley, 1983.

[Liskov 74] Barbara H. Liskov and S. N. Zilles. "Programming with Abstract Data Types," Proc. of the ACM Symp. on Very High Level Languages, _SIGPLAN Notices_, April 1974.

[Nelson 86] Robert W. Nelson. "NASA Ada Experiment -- Attitude Dynamic Simulator," _Proc. of the Washington Ada Symposium_, March, 1986.

[Parnas 72] David L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules," _Communications of the ACM_, December 1972.

[Rajlich 85] Vaclav Rajlich. "Paradigms for Design and Implementation in Ada," _Communications of the ACM_, July 1985.

[Seidewitz 85a] Ed Seidewitz. _Object Diagrams_, unpublished GSFC report, May 1985.

[Seidewitz 85b] Ed Seidewitz. _Some Principles of Object Oriented Design_, unpublished GSFC report, August 1985.

[Stark 86] Mike Stark. _Abstraction Analysis: From Structured Specification to Object-Oriented Design_, unpublished GSFC report, April 1986.

[Yourdon 79] Edward Yourdon and Larry L. Constantine. _Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design_, Prentice-Hall, 1979.

SESSION D.5	CAIS PANEL
Panel Chair:	David Pruett NASA Johnson Space Center
Panel members:	

Clyde Roby
Jack Krammer
Institute for Defense Analysis
Alexandria, Virginia

Sue LeGrand
SofTech
Houston, Texas

Robert Stevenson
Gould Electronics
Fort Lauderdale, Florida

Robert Fainter
Virginia Tech
Blacksburg, Virginia

Hal Hart
TRW Defense Systems Group
Redondo Beach, California