# An Application of Entity Life Modeling to Incremental Reengineering of Fortran Reactive Program Components

**F. G. Patterson, Jr.**
**Jean-Jacques Moortgat**

**Technical Report No. ISSE-TR-94-108**
**School of Information Technology and Engineering**
**George Mason University**
**Fairfax, Virginia 22032**

***Abstract.***

*This paper describes a case study in which the Entity Life Modeling (ELM) [1] technique is used to reconstruct a single module from a legacy FORTRAN application. The architecture of the module changed as a result of entity identification and mode analysis. Concurrent aspects of the problem domain were identified and modeled as concurrent tasks. To interface with the greater part of the software system that was not reconstructed, an interface object was created. The interface object may be regarded as scaffolding that will be discarded when neighboring increments of the system are reengineered. The reengineered increment has a highly maintainable design that is characteristic of complete systems constructed using the ELM methodology.*

## I. Introduction.

Reengineering is the process of reusing old programs to generate replacements. Traditionally, reengineering has meant modifying code that has only imperfect or incomplete specification-level and design-level information. Such maintenance includes: *perfective maintenance*, performed to realize new requirements level changes to the software product; *adaptive maintenance*, performed to allow software to continue to function in a new operational environment; and *corrective maintenance*, performed to remove the causes of software failures [2].

Reengineering begins with a decision on whether it is more economical to discard or to maintain old software that no longer meets the requirements of the users. The new environment may contain new hardware and software products that are incompatible with the old software. The user may have new requirements for additional functionality. It simply may be a propitious time for adding new capabilities. In any case the result is the same in that there is a need for adaptive and perfective maintenance on a program that is difficult to maintain. Before addressing new requirements, the old software may be analyzed and its components made reusable to support the required adaptive and perfective maintenance. We call this process *reconstructive maintenance*.

Reconstructive maintenance itself is neither perfective nor adaptive. Reconstructive maintenance effects a change in software development methodology which may fundamentally alter the software design architecture. A related concept is *restructuring*, which "actually changes the code by transforming ill-structured code into well-structured code[, focusing] solely on the source code" [3]. While this activity might be generalized to other life cycle activities [4], the concept is too restrictive for our purposes.

## Definitions of reengineering: narrow and broad.

**Reengineering**, also known as both *renovation* and *reclamation*, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [4]. The reengineering event, therefore, may be narrowly defined as the revision of the old software to allow necessary adaptive and perfective maintenance. In this way the old software is reengineered for immediate reuse. In this paper, this process is referred to as reconstructive maintenance. Success can be measured using reusability metrics and measures, such as those proposed by Fonash [5]. Broadly defined, reengineering includes all of these activities. The cost of reengineering that is to be compared to the cost of new development is not only the cost of reengineering the old software, but also the cost of the perfective and adaptive maintenance that is necessary to meet new requirements.

Reconstructive maintenance has a reverse engineering element and a forward engineering element. The core of reverse engineering technology is the abstraction of code into design concepts [6]. From this point we can both work backward to specification and work forward, through re-specification and redesign, to a better implementation [7, 8]. This can be difficult, however, when code has been fragmented through optimization or repeated maintenance, scattering clues to the design throughout the implementation. The cost of the reengineering activity depends partly upon the cost of the reverse engineering that is required during reconstructive maintenance.

Reconstructive maintenance will almost certainly be guided by some elements of adaptive maintenance, whether the source is to be reengineered incrementally or all at once. This is because a target environment will be chosen that not only facilitates interfaces to other software, but also supports a particular development methodology well, such as object-oriented design. For example, the new target environment may include a new programming language, a new graphical user interface, or a new host computer.

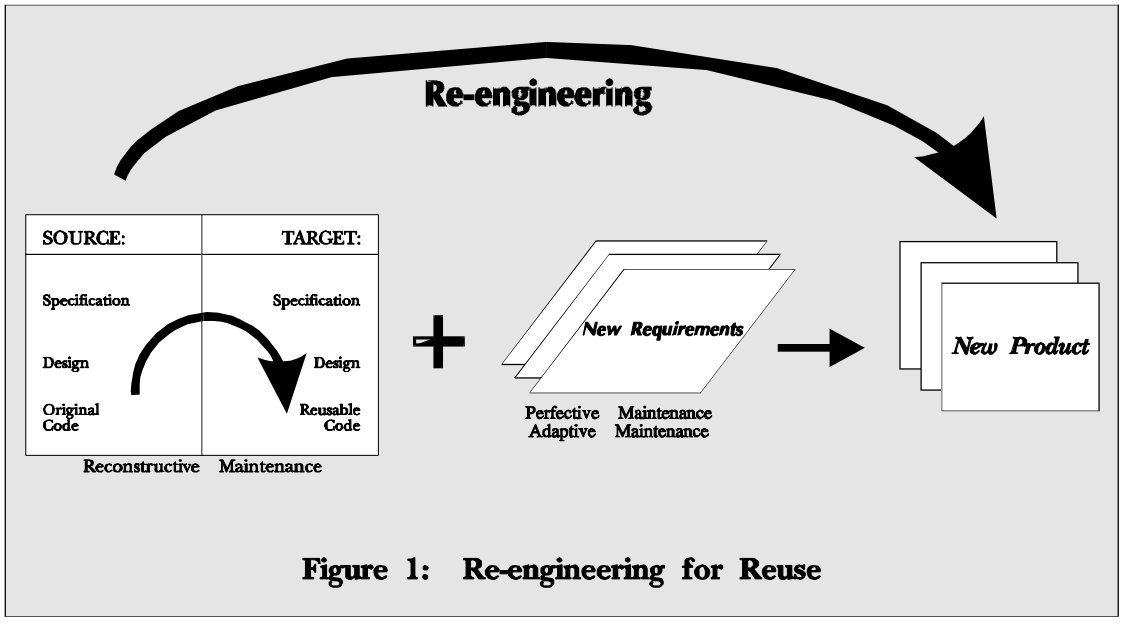


**Figure 1: Re-engineering for Reuse**

The overall reengineering process is depicted in Figure 1. The starting point is the *code* that is to be reengineered, shown on the left hand side of the Figure.

Information *may* already exist at the design and specification levels. If not, then some degree of reverse engineering may be necessary to provide complete information to support the reengineering activity. The finished product is the code on the right hand side, together with design and specification level documentation, upon which immediate perfective and adaptive maintenance may be performed.

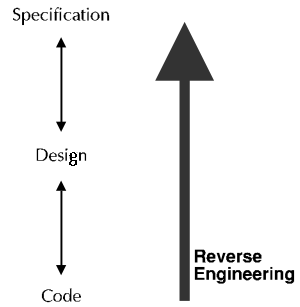

Figure 2

**Reverse Engineering** is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. The core of reverse engineering technology is the abstraction of code into design concepts. From the design point we can work upward to specification, as illustrated in Figure 2. To meet our needs, reverse engineering must be performed at least to the extent required to identify elements of our forward engineering model.

## Incremental Reengineering.

When it is not possible to reengineer an entire system, incremental reengineering may be an attractive solution that offers help in conserving both present and future resources. The selection of the increment is of critical importance in the management cost of the initial reengineering effort. Moreover, improper selection may decrease the level of maintainability of the software, leading to high maintenance cost.

There is a Pareto rule [9] that applies to software maintenance that says that 20% of the code is targeted for maintenance 80% of the time. Other things being equal, it is the critical 20% that should be reengineered to support a high level of

future perfective maintenance. On the other hand, there may be many other reasons for selecting one part of the code over another. For example: immediate and extensive maintenance may be pending for some part of the code; there may be known, but unresolved errors in some part of the code; or some part of the code is of outside origin (*e.g.*, is commercial off-the-shelf, or classified, or developed by others for another unrelated application and reused), and it is desired to reconstruct it for improved maintainability. Apart from these considerations, however, it is desirable to choose functions with few interfaces to reengineer.

Booch [10] lists quality factors in object-oriented methodology for interfaces that are consistent with *good* object-oriented design. There he states that a design should be loosely coupled.

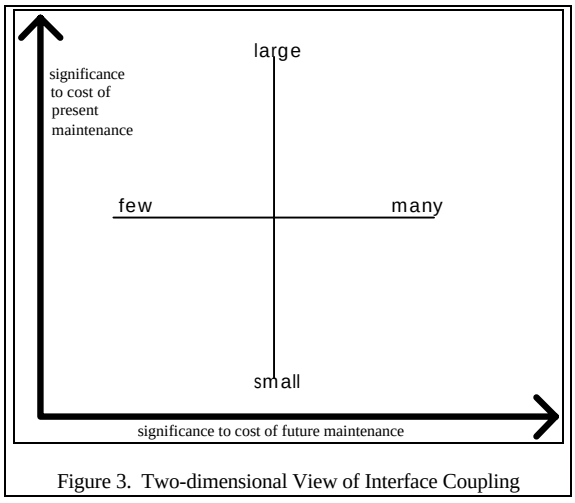

Figure 3. Two-dimensional View of Interface Coupling

As shown in the Figure 3, there is a two dimensional aspect to interface coupling. The number of interfaces may be few or many. The size of the interfaces may be small or large.

In the case when there are many interfaces, we can look at the two possible cases, *viz*, partial reengineering and total reengineering. First, if we are doing incremental reengineering, because each increment captures and repackages part of the function-oriented architecture, we tend to leave program interfaces intact among increments. In effect, there is a tendency to propagate dependencies, since most of the legacy architecture is retained. Second, in the case where reengineering of the entire software product is performed, a large number of interfaces tends to represent information about many objects. This translates into many interfaces in the object-oriented design. Thus, whether incremental or total reengineering is performed, a large number of interfaces in the function-oriented design tends to create a large number of interfaces in the object-oriented design [11].

Furthermore, a large number of interfaces in the software means that the software will be more difficult to change since there are more potential side effects. This is indicative of low adaptability. According to Sommerville, "For optimum adaptability, a component should be self-contained. A component may be loosely coupled in that it only cooperates with other components *via* message passing. This is not the same as being self-contained as the component may rely on other components, such as system functions or error handling functions. Adaptations to the component may involve changing parts of the component which rely on external functions so the specification of these external functions must also be considered by the modifier.... To be completely self-contained, a component should not use other components which are externally defined. However, this is contrary to good practice, which suggests that existing components should be reused" [12].

There are two aspects of coupling in the FORTRAN programs that have been examined in this work. One is the list of parameters passed between subroutines. The other is the variables stored in FORTRAN COMMON blocks that are shared between subroutines. The interface between any two subroutines, consisting of passed parameters and global variables, may be arbitrarily large and complex, depending upon the number of variables and the complexity of their data types involved in the interface. As a subordinate theme of this paper, we assert that there should be *few* interfaces and *small* interfaces among objects in our reengineered product. By few interfaces, we mean that the reengineered increment exchanges data with a small number of other program units. By small interfaces, we refer to the number of variables and the complexity of their types in a given interface. In reaching this goal, we believe that we can apply Booch's criterion to the functions that we select for encapsulation in a newly formed object in our reconstruction effort.

## II. Case Study: Application of ELM to the GOFOR System.

Entity-Life Modeling is a method of software engineering that has elements in common with both function-oriented and object-oriented methods. As in object-oriented design methods, the first step is the identification of objects from the problem domain, the identification of object attributes and operations belonging to each object, and the design of class structures that encapsulate state information and export attributes to other objects as needed. ELM departs from object-oriented methods in its ability to manage the timing and ordering of events as in some function-oriented methods. Threads of execution are defined wherein *entities* exhibit sequential behavior by operating on objects, perhaps concurrently with other entities.

The application of ELM to reengineering involves the identification of entities and their threads of execution just as in forward engineering complete systems. Some reverse engineering may be necessary to identify objects and their behaviors.

The steps in the application of ELM to incremental reengineering may be listed as follows:

1. Identification of entities
2. Identification of concurrent tasks
3. Creation of Buhr diagrams
4. Design of interface objects
5. Composition of state transition diagrams

In general it may be necessary to reverse engineer the legacy code to accomplish steps 1, 2, and 5. Steps 1, 2, 3, and 5 represent the steps that are customarily used in the ELM technique. We added step 4, necessary to allow the reengineered increment to communicate with the remaining legacy software.

**The GOFOR System.**

For this case study the NASA Goddard Space Flight Center (GSFC) furnished a copy of a FORTRAN system which was developed using a function-oriented development model. We received a copy of the Geostationary Operational Environment Satellite-I (GOES-I) Simulator Support in FORTRAN (GOFOR) software. GOFOR is the attitude dynamics and control system part of the much larger GOES Simulation System (GSS). We were given the GOFOR code on a diskette, together with the Detailed Design Document. We also received telephone access to a programmer at GSFC who is familiar with the system.

We reduced the amount of code we attempted to restructure by looking only into a single subsystem of the GOFOR system, the Attitude and Orbital Control Subsystem (AOCS), and choosing part of the subsystem. The AOCS was chosen for several reasons: (1) It is one of the two "deliverable" subsystems in the software model; the others may be regarded as scaffolding software needed to test the two deliverable subsystems. (2) It is the most complex and the most changed over its life cycle to date; therefore, the benefits from increasing the reusability of this program unit will tend to amortize the reengineering cost more quickly. (3) The interfaces to the AOCS are limited to one other subsystem, although the interface is not small, since the two subsystems share a large number of Common block variables. In addition, we noted several subroutines with more than ten formal parameters required for normal operation. (4) The GOFOR Detailed Design Document (DDD) has a reasonable level of documentation available to assist in understanding the relationships among the program units that compose the AOCS. This greatly assisted our reverse engineering effort.

The GOFOR program code is composed of thirty files, containing a main program, twenty-six subroutines, and three "include" files containing data structure definitions, all written in the FORTRAN 77 programming language. The system was developed on and designed to run on a Vax 11/780 computer at GSFC. Two sources of program documentation were supplied with the program code. They are the "GOFOR Detailed Design Document" and the internal program documentation in the form of a prologue of comment statements at the beginning of each file.

The first step in incremental reconstructive maintenance is the identification of the subset [13] of the program that will be affected and the identification of the objects within that subset. We intended to reconstruct a subsystem from the GOFOR software system and to record the origins of global data and its relationship to the objects that are chosen for the reconstructed object-oriented design. First, we intended to reverse engineer from the FORTRAN code to identify COMMON blocks and their contents and the role of each data element found. In addition, other sources of global data were to be identified and analyzed. The next step was the forward engineering step of identifying the new object's attributes and operations.

The GOFOR system is functionally divided into five subsystems as follows:

— Truth Model (TM) which contains the integration of the rigid body dynamics and all attitude sensor and actuator models

— The Attitude and Orbit Control Subsystem (AOCS) which contains the control logic necessary to duplicate each spacecraft control mode and also represents the logic transitions between various control modes. AOCS receives attitude sensor output from the TM and sends thruster commands to the TM.

— The Profile Program (PP) which models the environment of the spacecraft and pre-calculates a profile data set for use by the simulator.

— The Simulation Driver (SD) is the execution driver for the simulator. It accepts user input, controls and manages the execution, and generates output data file for plot generation.

— The Plot Generator (PG) is run after simulation is completed to generate printer and CRT plots and tabular reports.

The GOFOR Detailed Design Document (DDD).

GSFC furnished a *preliminary* version of the DDD, which contains an introduction, an overview of the system, and a description of five "subsystems," each of which comprises a proper subset of the thirty program files. Each subsystem design description provides a list of the major functions in the subsystem, a high-level interface block diagram, a structure chart, a high-level description of each subsystem component, and a list of the subroutines involved. Since publishing the document, which was never revised, the code has undergone extensive modifications, including the addition of entire modules and the addition of new functions and new data.

As a practical matter of procedure, GSFC defines FORTRAN COMMON blocks in *include* (".INC") files. Three ".INC" files were included in the source code that we obtained. We quickly noted that more than three file names were *include*d in some of the subroutines. Using the UNIX tool *grep*, we listed out all of the *include* statements in the GOFOR system and counted twenty-nine such files.

We found that we were able to reverse engineer at a high level the program code, and to find correspondence to the external documentation, to provide a satisfactory preliminary picture, in the form of a high level, annotated DFD, of *what processes* pass *what data* and *when*.

Description of the AOCS.

The AOCS is a collection of FORTRAN subroutines and COMMON block data elements that perform the function shown in Text Box 1.

```
C   METHOD:
C   -------
C
C     CREATE COMMONS
C     DECLARE VARIABLES
C
C     CALL SENPRO TO PROCESS ATTITUDE SENSOR DATA
C     CALL CONTRL TO SELECT AOCS CONTROL MODE
C     CALL ATTCON TO PROCESS CONTROL LAW FOR ASSOCIATED MODE
C     CALL VCOILD TO PROCESS THRUSTER COMMAND GENERATION
C
C     RETURN
C     END
```

**Text Box 1**

Subroutines SENPRO, CONTRL, ATTCON, and VCOILD are all called by subroutine AOCS. Each of these four subroutines, in turn, calls other subroutines. The flow of control is controlled by the control mode of the AOCS and the attitude sensor data. The control mode is, in turn, controlled by the attitude sensor data and by the user. The output from AOCS is a set of commands and settings for thrusters, magnetic torquers, and momentum wheels. Reorganizing this information and abstracting a bit, we can view the AOCS interface [14] to the Truth Module Subsystem (TM) as shown in Figure 4.

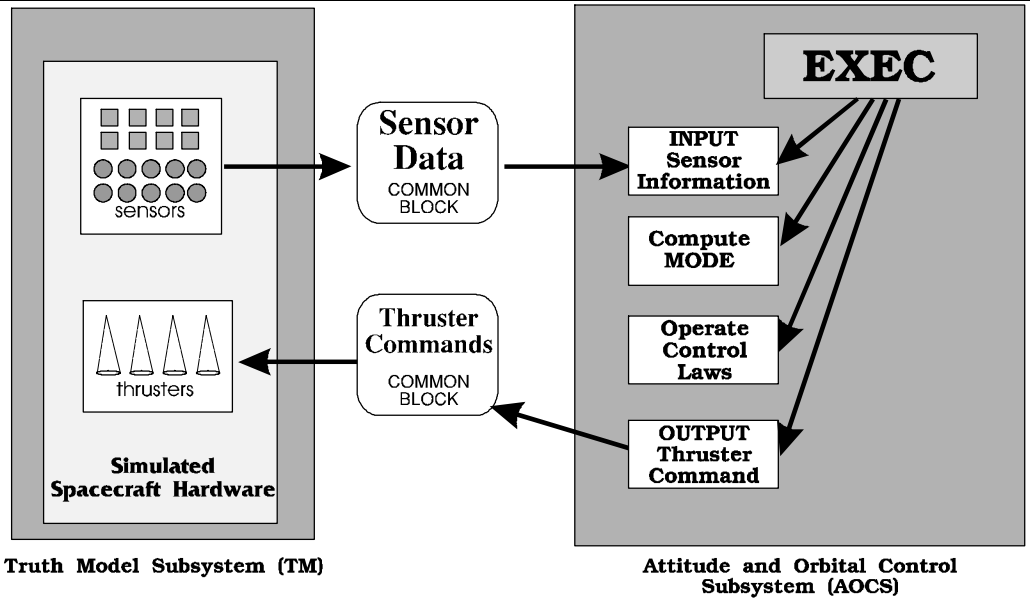


**Figure 4**

Description of the interface.

In terms of the software, the AOCS interfaces only with TM. This is a total of one interface (two if input and output are counted separately). Thus, our criterion of bounding an increment with a *small number* of interfaces is satisfied. In terms of the "real world" objects represented by the software, however, the TM is not a single interface, since the attributes of many objects on board the space craft are being simulated. For example, the input interface (Sensor Data COMMON block) represents five types of sensors, each more useful in some operational modes than in others. Since modes are very important to AOCS, a real world view of TM would show five input interfaces. We took the view that the software *is* the real world for the purposes of incremental reengineering. Only a full reengineering of the code in its entirety would reverse engineer back into the reality. Thus, we treated the interface as a single complex input into AOCS from TM. Likewise, the types and numbers of thrusters that are commanded by the AOCS represent an increase in the number of elements in a single interface (*i.e.*, the size of the interface), not an increase in the number of interfaces.

The AOCS Super-Object.

The decision to reengineer incrementally, limiting the scope to the AOCS subsystem, requires that the AOCS Object be functionally equivalent to the AOCS Subsystem that it replaces (see right-hand side of Figures 4 and 5). Moreover, until such time that further incremental reengineering is performed, the AOCS Object must interface with its software environment in exactly the same way as the AOCS Subsystem that it replaces. The primary environment for AOCS is supplied by TM and by FORTRAN COMMON blocks; therefore, the AOCS object must interface to its environment in the same way through COMMON blocks and parameters. Internally, the AOCS object may be redesigned as necessary.

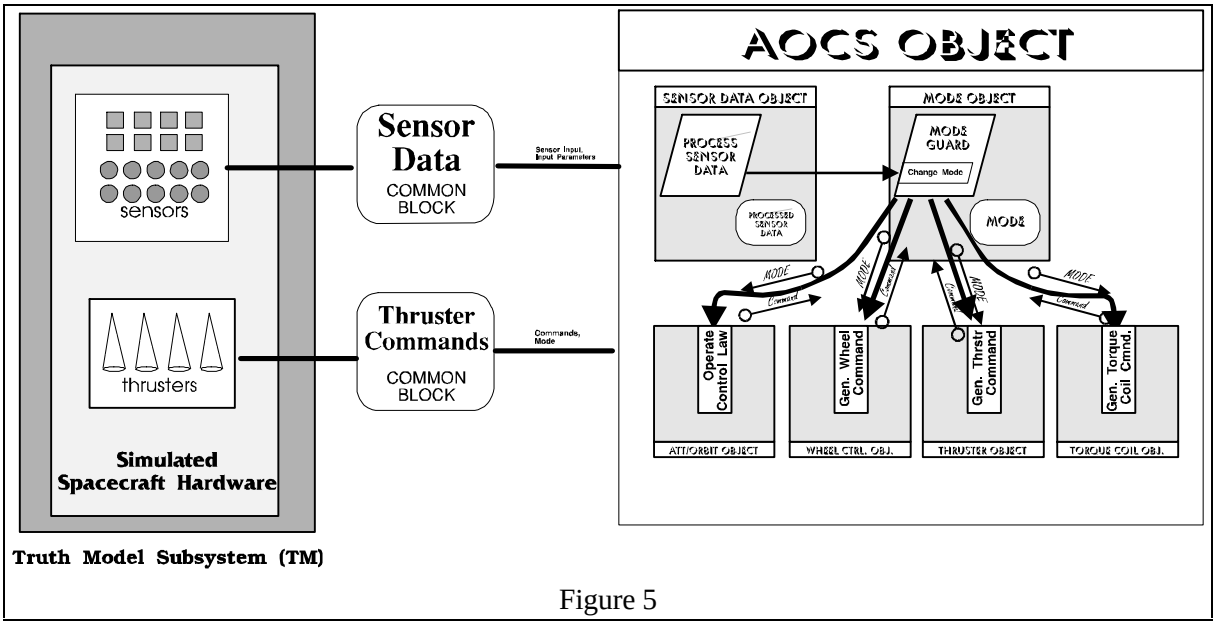


Figure 5

The AOCS object is an interface to the rest of the program, and it is a boundary around a set of objects: a super-object. This is essentially the same as the role played by the function-oriented AOCS subsystem, which is a boundary around a collection of FORTRAN subroutines. The ideas are different, however, in that the AOCS object may be eliminated when reengineering of the entire GOFOR system is complete, since the AOCS object is scaffolding for a previous increment and which is no longer needed. Thus, the purpose of the AOCS object is to provide the interface to the function-oriented remainder of the GOFOR software. Its function is to maintain the state of the interface and to transmit and receive information to and from the interface.

A global table was prepared that contained most of the elements of the interface found in FORTRAN COMMON blocks. The table was created, using UNIX *grep*, from comment statements in the FORTRAN code. The common blocks were left intact, since, as noted above, the interface was out-of-scope for this incremental reengineering effort. However, some of the elements of the interface were folded into objects within the AOCS super-object as attributes of those objects.

The AOCS Objects.

The AOCS structure charts were analyzed to determine the architecture, functionality, and control structure of each of the internal modules. The AOCS is mode driven, in that each mode has specialized functions to be performed, based upon certain sensor information, and based upon attitude control laws and thruster control laws associated with each mode. Thruster commands are output based upon the inputs and the mode. Most of the inputs were documented in the global table described above. AOCS subroutine parameters are used to initialize the AOCS. We designed the AOCS object as shown in the high level Buhr [1, 15] diagram in Figure 5.

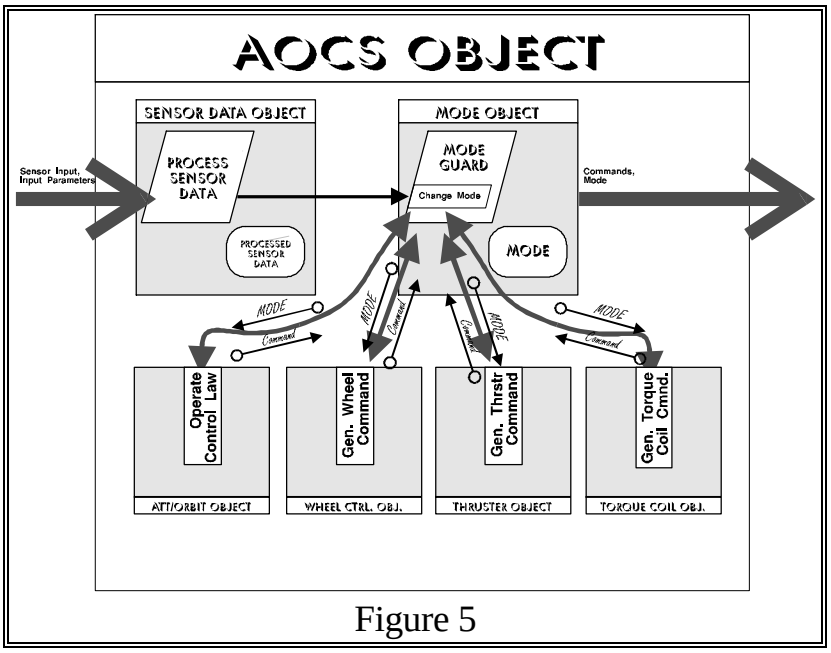


Figure 5

Each object in the AOCS super-object is the owner of a portion of the data that was previously found only at the interface between AOCS and TM. In our design, an input task, Sensor Data Object, continuously reads the input data and computes limit data and other parameters that are need by the other objects in the AOCS. The Sensor Data Object can detect error conditions in the sensor readings. When problems are detected, the Sensor Data Object can determine that the mode of operation needs to be changed and send a message to the Mode Object which, in turn, changes the mode of operation accordingly. We modeled the Mode Guard as a task, since mutual exclusion to this critical variable is important. There are eight modes of operation identified in the DDD according to whether the spacecraft is monitoring the earth or the sun, is in safe mode, is out of contact with the signal from the ground control, *etc*. In each mode of operation, the four objects behave in a mode-specific way. An example is included in Appendix A. We developed state transition diagrams based on the analysis of the modes of operation.

Encapsulation of interface information.

Each of the four objects encapsulates state information, control law information, and a command generator for making adjustments to the flight characteristics of the spacecraft. To test our hypothesis that COMMON block information in the function-oriented design will be allocated to objects in the object-oriented design, we used the common block descriptions information found in the DDD. For each variable in the list, we were able to add the variable to one of three lists: (1) variables (attributes) owned by an object in AOCS; (2) variables owned by TM, usually simulated hardware objects; or (3) variables that were not allocated to list 1 or list 2.

> List 1. As a result of analysis, list 1 contained primarily commands bound for the simulated magnetic torquer, the momentum wheel, or the thrusters. These commands were generated by subroutines in the original module AOCS and were written into common blocks for the consumption of TM. This is reasonable, since TM contains all of the (simulated) commandable hardware. We were able to allocate all of these commands to objects in our object-oriented design.
>
> List 2. Sensor readings and commands from ground control were allocable to this list, since the readings and commands are attributes of hardware and ground objects simulated in TM.
>
> List 3. List three contained problems that were difficult to solve until it was observed that if the unidentified parameters are not found upon searching the AOCS and TM source code, then the parameters must belong to the simulation driver (SD) or some other module in the larger system. In a complete object-oriented reconstruction of the system, these interface objects would be part of some object in some other part of the architecture. The point is that the items on List 3 are not part of the TM-to-AOCS interface at all and are, therefore, out-of-scope.

We were able affirmatively to answer the question of whether a single very large, complex, heterogeneous interface was entirely allocable to objects in the

object-oriented redesign. Large interfaces tend to become internalized as attributes of program objects and may or may not be transmitted across object boundaries. Because of this, reengineering of future increments will not need to deal with the large interface. On the other hand, the number of interfaces is very important in incremental reconstruction, since interface objects must be constructed for every interface to the reconstructed portion of the software.

## III. Conclusions.

1. *The choice of increment greatly affects the product*. In bounding an increment to reconstruct, we were mindful of two costs: the cost of the reconstruction activity itself (this increment) and the cost of future reconstructive maintenance (future increments). We noted in this paper several factors that influence the choice.

   First, the "80-20 rule" might add cost to present reconstruction for any of a number of reasons. For example, the code that usually is needful of maintenance may be highly complex, either inherently or because repeated maintenance has obfuscated the semantics of the code; the code may be tightly coupled with other code with potential maintenance side effects; or the code may be the implementation of a poor design or a poor requirements specification. In all of these cases, we believe that the high present cost of reconstruction is a good investment in that future maintenance of all kinds will be more successful and less costly.

   Secondly, we noted that the code that is already scheduled for adaptive, perfective, or corrective maintenance might be included in the present increment. This has the advantage of reducing the cost of the scheduled maintenance, thus partially amortizing the cost of reconstruction beginning immediately. However the increment should be highly cohesive. Otherwise, the effect is to reconstruct two more or less unrelated increments.

   Thirdly, it was noted that the increment to be reconstructed should have interfaces to few program units outside its boundary. This is because the size and complexity of the interface object that must be created to connect the reconstructed increment to the remainder of the system is largely a function of the number of interfaces that must be established. A second reason is related to the potential for causing unwanted side effects in the remainder of the system as a result of future maintenance activities. It was noted that the size of these few interfaces was largely irrelevant to the cost of future maintenance, although more expensive in terms of building the

interface object, since the elements of the interface will be partitioned into object attributes and encapsulated within object boundaries in the reconstructed increment.

2. *The ELM method is modified for reconstruction by the addition of the construction of the interface object.* To interface with the greater part of the software system that was not reconstructed, an interface object was created. The interface object may be regarded as scaffolding that will be discarded when neighboring increments of the system are reengineered. Thus, the interface object performs a secondary maintenance function by acting as a road map for reengineering the next increment.

3. *The architecture of the reengineered increment is different from its predecessor* as a result of entity identification and mode analysis. We believe that ELM provided a degree of fidelity to the problem domain that was not achieved by the structured design method originally used to develop GOFOR. ELM lends itself well to a problem such as AOCS where there are well defined entities, well defined processes that affect the entities, and aspects of processing that are concurrent.

   Additionally, ELM enabled us to design a sampler task for the sensors and a guardian task for the mode control. ELM's State Transition Diagrams depicted the attitude control modes in a way that's closer to how the spacecraft attitude control actually works (*i.e.*, events occur that activate modes depending on the spacecraft's location or attitude). ELM's Buhr diagrams help us design concurrent aspects of the problem such as the sensor sampler and mode guardian tasks that we have described.

4. *To accomplish mode analysis, it was necessary to go to the code.* The structure charts from the DDD and the text associated with it did not provide enough detail to develop the state transition diagrams or the Buhr diagram for our reengineered design. For example, Text Box 2 presents source code from the AOCS system. This type of code appears in each of the current mode subroutines for AOCS. The subroutines are reading from COMMON blocks and using data from the sensor processing routine to determine if transition is

needed from one mode to another (*e.g.*, to Sun Acquisition Mode). Condition statements are used to check several different flags that are passed to the routine. Based on these flags, which provide information on the spacecraft's position, and on control laws, automatic transition occurs from one state to another. Considerable analysis of the source code enabled us to find threads in the processing which we reengineered into state transition diagrams. In doing so we improved our understanding of how to group processes with the data that affects them.

5. *To accomplish object identification it was necessary to go to the interfaces and partition them into candidate objects.* As noted above, AOCS had a single interface to external code, the interface with TM. The single interface was accomplished through several named FORTRAN COMMON blocks which contained information about the objects in the problem domain. The elements of the interface were partitioned as candidates for objects. This technique was carried out and validated as a technique for finding objects for the entire GOFOR code in another effort.

6. *To identify entities it was necessary to reverse engineer back to requirements.* Although we were able to identify objects through examining the interface and to relate functions to objects by examination of the code, we do not believe that candidate entities could have been identified without an understanding of the problem domain. Our minimal understanding of the sensors and effector hardware upon which the simulation was based was the principal data that we drew upon in identifying concurrent sequential behavior.

```
C
C***  ORMODEL TWO AUTOMATIC MODE TRANSITIONS
C
      IF (.NOT. (SAFMOD .OR. (SAFMOD_F .AND. AUTSAF))) THEN
            IF (INIT_SUNACQ) THEN
                  MODE = 3
            ENDIF
C
      IF (XSIT_CMPLET) MODE = 5
      IF (AEARTH) THEN
            MODE = 4
      ENDIF
C
      IF (MODE .EQ. 4) THEN ! EARTH ACQUISITION
C
            IF (NO_PREVIOUS_EARTH_PRESENCE) THEN
C  NO PREVIOUS EARTH SIGNAL
C
                  IF (EARTH(1) .OR. EARTH(2)) THEN
                        NO_PREVIOUS_EARTH_PRESENCE = .FALSE.
                  ELSE
C
C  EARTH ACQ CMDED BUT NO EARTH PRESENT
C  FOR ROLL EARTH ACQUISITION
C
                  IF (.NOT. EARTH(2).AND. .NOT. EARTH(2)
                                    .AND. SUBMODE .EQ. 0) THEN
                        NEARTH = .TRUE.
                        AEARTH = .FALSE.
                        INIT_SUNACQ = .FALSE.
C  NORMAL SUN ACQUISITION MODES
                        MODE = 3
C  AUTO TRANSITION TO SUN ACQUISITION
```

**Text Box 2**

Summary.

The result was a product that

a. encapsulated state information in objects, reducing coupling,
b. realized timing requirements of the original code,
c. identified and utilized opportunities for concurrency, and
d. modeled in software with high fidelity the hardware entities in the problem domain, thus yielding a high level of cohesion.

## APPENDIX A.

The following example shows how processing is dependent upon the current mode of the system.

**AOCS MODES:**
- **RATE DAMPING MODE**
- **SUN ACQUISITION MODE**
- **EARTH ACQUISITION MODE**
- **NORMAL ON-ORBIT MODE**
- **STATION-KEEPING MODE**
- **BACK-UP STATION-KEEPING MODE**
- **STATION-KEEPING TRANSITION MODE**
- **SAFE MODE**

As an example, we have done the following analysis for the Earth Acquisition mode (EARACQ). This mode occurs when the spacecraft's position is incorrect in relation to the Earth. An incorrect position is recorded by the Earth Sensor Assemblies (ESA), CASSA and DSSA and the spacecraft rate is recorded with the DIRA sensor. The recording of this sensor information is modeled in the Truth Model (TM). This information is sent to the AOCS. The AOCS goes into an earth acquisition mode to correct the position and rate.

The following arguments are passed to EARACQ:

```
ARGUMENT LIST:
ARGUMENT I/O TYPE DIM DESCRIPTION
-------- --- ---- --- -----------
IDDNUM    I  I*4    1  LOGICAL UNIT NUMBER FOR DEBUG
IDDWRT    I   "     1  DEBUG FLAG
TCYCL     I  R*8    1  AOCS CYCLE TIME (SEC)
CASANG    I  R*4    3  CASSA ANGLES (ROLL/PITCH/YAW)
ESANG     I   "     2  ESA ANGLES (1 : ROLL;  2 : PITCH)
DSANG     I   "     2  DSS ANGLES (2 : ALFA;  1 : BETA)
DIRANG    I   "     3  DIRA ANGLES (ROLL/PITCH/YAW)
DIRRAT    I   "     3  DIRA RATES (ROLL/PITCH/YAW)
IOUT      O  I*4    3  THRUSTER COMMAND PULSE
EARTH     I  L*1    2  EARTH PRESENCE FLAG
WSPEED    I  R*4    3  WHEEL SPEEDS (2 MOM., 1 REC)
WHLSAF    I  L*1    1  WHEELS IN SAFE MODE
WHLVLT    O  R*4    3  WHEEL TORQUE COMMAND VOLTAGE
```

The logic is as follows:

```
EARTH ACQUISITION AXIS CONTROL LOOPS:
COMPUTE INPUT POSITION ERROR BASED ON POSITION SWITCHES
DO FOR EACH AXIS
      CALCULATE FILTERED BIAS-CORRECTED POSITION ERROR
      CALL LIMITR TO FILTERIZE BIAS-CORRECTED POSITION ERROR
      CALCULATE FILTERED BIAS-CORRECTED RATE FROM DIRA RATE
      COMPUTE FINAL POSITION & RATE ERROR COMMAND
      CALL DBAND TO CHECK WHETHER OR NOT SIGNAL IS WITHIN
                                        DEADBAND REGION
      CALL PWPF TO PRODUCE THRUSTER CONTROL COMMAND
END DO FOR
CALL WHLCTL
```

Additionally, the position switches and rate biases are set relative to the positioning of the spacecraft as seen below:

```
EARTH ACQUISITION PHASE CONTROL:
DO CASE - SUBMODE
      CASE 1 : ROLL EARTH ACQUISITION
      SET POSITION SWITCHES

      CASE 2 : PITCH/YAW CALIBRATION
      SET POSITION SWITCHES

      CASE 3 : DIRA ATTITUDE REFERENCE
            SET POSITION SWITCHES

      CASE 4 : PITCH EARTH ACQUISITION
      SET POSITION SWITCHES
END DO CASE
```

# References


[1] B. Sanden, *Software Systems Construction with Examples in Ada*. Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[2] E. B. Swanson, "The Dimensions of Maintenance," *Proc. 2nd International Conf. on Software Engineering*, vol. IEEE Catalog No. 76CH1125 4 C, pp. 492-497, 1976.

[3] S. L. Pfleeger, *Software engineering: the production of quality software*, 2nd ed. New York: Macmillan Publishing Company, 1991.

[4] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, pp. 11-12, 1990.

[5] P. M. Fonash, "Metrics for Reusable Software Code Components," Ph.D. Dissertation, George Mason University (Fairfax, Virginia), 1993.

[6] R. N. Britcher and J. J. Craig, "Using Modern Design Practices to Update Aging Software Systems," *IEEE Software*, vol. 3, pp. 16-24, 1986.

[7] G. W. Jones, *Software Engineering*. New York: John Wiley & Sons, Inc., 1990.

[8] V. A. Berzins and Luqi, *Software engineering with abstractions*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1991.

[9] H. M. Sneed, "Economics of Software Re-engineering," *Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Ltd.*, vol. 3, pp. 163-182, 1991.

[10] G. Booch, *Software engineering with Ada*. Redwood City, California: Benjamin/Cummings, 1987.

[11] B. Meyer, *Object-Oriented Software Construction*. New York: Prentice Hall, 1988.

[12] I. Sommerville, *Software Engineering*, 4th ed. Wokingham, England: Addison-Wesley Publishing Company, Inc., 1992.

[13] E. S. Garnett and J. A. Mariani, "Software reclamation," *Software Engineering Journal*, pp. 185-191, May. 1990.

[14] I. Jacobson and F. Lindstrom, "Re-engineering of old systems to an object-oriented architecture," *Proc. OOPSLA-91*, pp. 340-350, 1991.

[15] R. J. A. Buhr, *Practical Visual Techniques in System Design: with Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.