N92-22471

REDUCING THE COMPLEXITY OF THE SOFTWARE DESIGN PROCESS WITH OBJECT-ORIENTED DESIGN

M. P. Schuler
(804) 864-6732
NASA Langley Research Center
Hampton, VA 23665-5225

ABSTRACT

Designing software is a complex process. The purpose of this paper is to describe and illustrate how Object-Oriented Design (OOD), coupled with formalized documentation and tailored object diagraming techniques, can reduce the complexity of the software design process. The OOD methodology described uses a hierarchical decomposition approach in which parent objects are decomposed into layers of lower level child objects. A method of tracking the assignment of requirements to design components is also included. Increases in the reusability, portability and maintainability of the resulting products will also be discussed. This method was built on a combination of existing technology, teaching experience, consulting experience, and feedback from design method users [1] [3]. The concepts discussed in this paper are applicable to hierarchal OOD processes in general. Emphasis will be placed on improving the design process by documenting the details of the procedures involved and incorporating improvements into those procedures as they are developed.

INTRODUCTION

A simplified version of an actual project design, for a distributed dynamic controls system, will be used as a case study in describing the OOD process. The controls system was required to: obtain inputs from analog sensor devices attached to a large structure; convert those inputs into digital form; calculate actuator output commands based on the sensor inputs; perform a digital to analog conversion on the actuator commands and send those analog commands to actuators connected to the structure. The intended outcome of this closed loop process was to control the structures movement However, the design examples used for illustration will primarily be concerned with the subsystem responsible for system configuration and data recording, since it does not require a detailed understanding of the application domain. Figure 1 defines the design symbols which will be used in the examples.

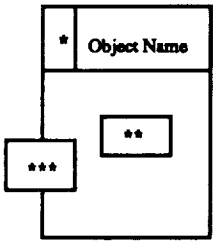
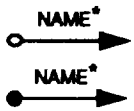
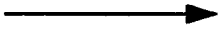
	This symbol represents an object. * This rectangle is inserted if the object is either a terminal or external object. A terminal object is an object that cannot be decomposed any further since each requirement assigned to it is executed by the object itself or its operations. If the object is terminal, a T is placed in this rectangle. An external object is an object which is allocated to a physically separate subsystem and is represented by placing an E in this rectangle. ** Inside this rectangle is the name of an operation that is internal to the object. *** Inside this rectangle is the list of callable operations on the object.
	'Data Couple' is the technical term used for this symbol. *If the circle of the tail of the arrow is empty, the NAME represents the data flow associated with the data couple. If the circle is solid, it represents the command associated with the data couple.
	This symbol is a control arrow. The object at the tail of the arrow controls the object at the head of the arrow.
	This symbol represents a data source or a data sink.

Figure 1. Basic symbol definitions.

This paper focuses on the preliminary design phase. It is assumed that prior to this phase a thorough requirements analysis has been performed and a software requirements document has been completed. The analysis results and the software requirements document are the input documents to the preliminary design phase.

An important goal from the start of this design was to partition the modules of the support domains from those of the application domain (the dynamics controls domain). In other words, to design the system so that code modules produced for different domains would be loosely coupled. This would reduce the complexity of the design and also produce products that were highly reusable, portable, and maintainable.

PRELIMINARY DESIGN DECOMPOSITION STEPS

During the preliminary design phase, a step-by-step process for object identification and decomposition was applied iteratively. For discussion purposes, the object being decomposed will be called a parent object. The objects it is decomposed into will be called the child objects. The following provides a description of each of the steps (Figure 2).

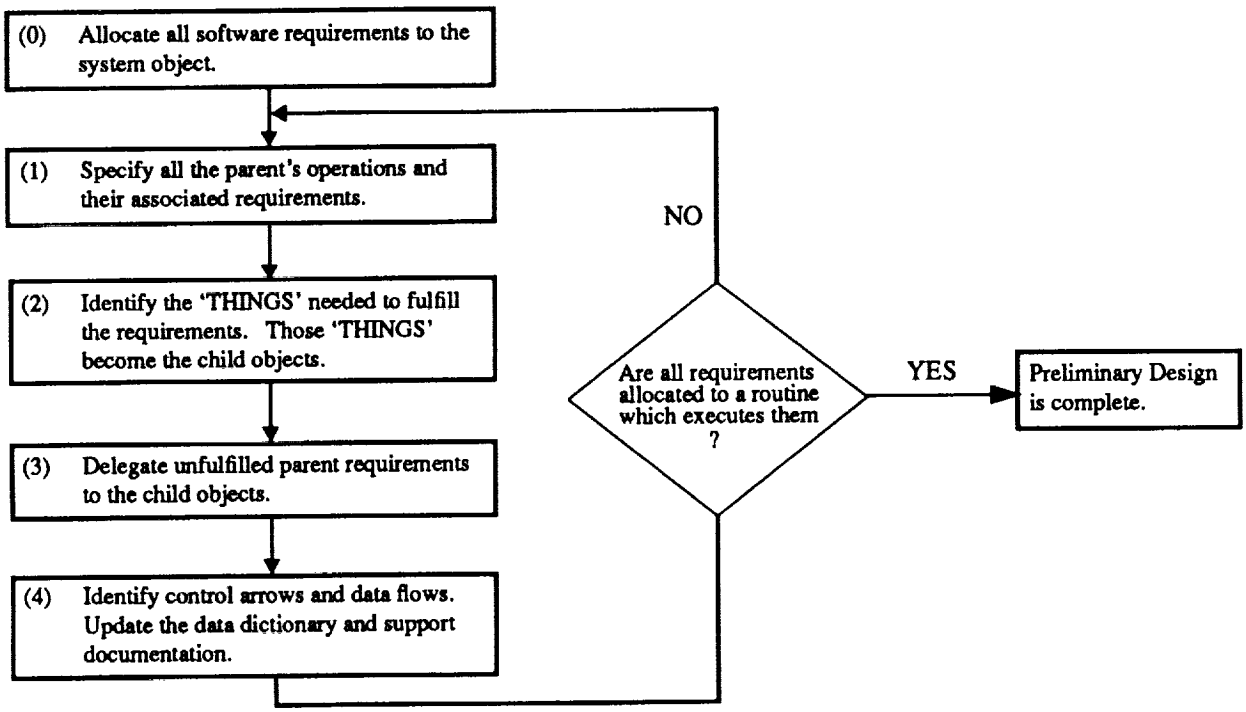


Figure 2. Preliminary design decomposition steps.

(0) Allocate all software requirements to the system object. This is the parent object to the first level of the system's decomposition. This step is done only once.

(1) Review the parent object's requirements and identify all the parent's operations. All requirements must be fulfilled by the parent itself or assigned to one of the parent object's operations. However, those operations may not execute all of the requirements, since during decomposition many of the requirements may be delegated to, and fulfilled by, the child objects or their operations. A textual description, along with the allocated requirements for each of the operations, is recorded on an Operations Description form (Figure 3).

(2) Identify all the child objects. Step through the parent object's requirements, in the order in which they would be executed. The purpose is to determine the primary THINGS needed to fulfill the requirements. In other words,

walk through what needs to be done to determine what THINGS are needed to do it. For each of those THINGS a child object is created thus defining the parent object's decomposition. A textual description of each child object is recorded in the Object Description forms (Figure 3).

(3) All unfulfilled requirements from the parent object are decomposed and assigned to the child objects. All assignments are recorded in the Object Description forms.

(4) Control arrows and data flows between objects are identified and diagramed. A data dictionary is updated and an Object or Operation Description is completed for each element of the design.(Specifying the detailed control and data flow between objects helps identify operations as well as reduce the subjective nature of the object design diagrams.)

OBJECT DESCRIPTION	OPERATION DESCRIPTION
NAME: Specify the object name and library number.	NAME: Specify the operation name and library number.
VERSION NUMBER / DATE: This number and date is updated each time the description is updated.	OJBECT: Specify parent object name and library number.
DESCRIPTION: A brief written description of what the object is required to do.	VERSION NUMBER / DATE: This number and date is updated each time the description is updated.
REQUIREMENTS: Specify the requirements allocated to this object.	DESCRIPTION: A brief written description of what the operation is required to do.
OPERATIONS AND PARAMETERS: Callable operations on this object.	REQUIREMENTS: Specify the requirements allocated to this operation.
ASSUMPTIONS: List assumptions made concerning those things needed to fulfill this objects requirements.	PARAMETERS AND TYPE: Specify the parameters and types if known.
INTERNAL INFORMATION: Specify internal objects and operations.	EXCEPTIONS: List all exceptions identified thus far.
ISSUES: Unknowns that must be determined before this object description can be considered complete.	ASSUMPTIONS: List assumptions made concerning those things needed to fulfill this operations requirements.
	ALGORITHM: Give the algorithm/pseudocode specifying what the operation will do to fulfill its requirements.

Figure 3. Object and Operation Description Forms.

Steps 1 through 4 are repeated until all system requirements have been allocated to an object or operation which executes them. Requirements allocation is a two-step process. In step 1, all the requirements not executed by the parent are allocated to the parent's operations. In step 3, requirements that were not fulfilled by the parent or its operations are decomposed and allocated to the child objects. If a child object is not terminal[1], it then becomes a parent object and is decomposed. To assure that each requirement is executed by some part of the design, a requirements traceability matrix is constructed. The matrix traces the correspondence between the requirements and

1. A terminal object is an object that cannot be decomposed any further since each requirement assigned to it is executed by the object itself or its operations.

the objects or operations that execute them. Assuring the traceability of requirements to the design is achieved by verifying that: all requirements are listed in the matrix; that an object or operation is assigned to fulfill each requirement; and that those requirements are specified in the description forms.

PRELIMINARY DESIGN

Dynamic_Controls_System Object Decomposition

The first object to be defined in the preliminary design was the Dynamic_Controls_System (Figure 4), which represented the system in its entirety. All software requirements were delegated to this system object. It was then decomposed into three child objects, one for each of the computer subsystems specified in the requirements. The System_Manager was one of the three child objects defined at this level. The other two child objects will be referred to as Subsystem_One and Subsystem_Two. An Object Description form was drafted for each of the objects defined thus far. The form was used to capture all the available information about an object and therefore included a detailed textual description of this level of decomposition (Figure 3). A brief description defining what each object is required to do was included. All the system requirements were broken down and assigned to the three child objects. These assignments were also recorded in the Object Description forms.

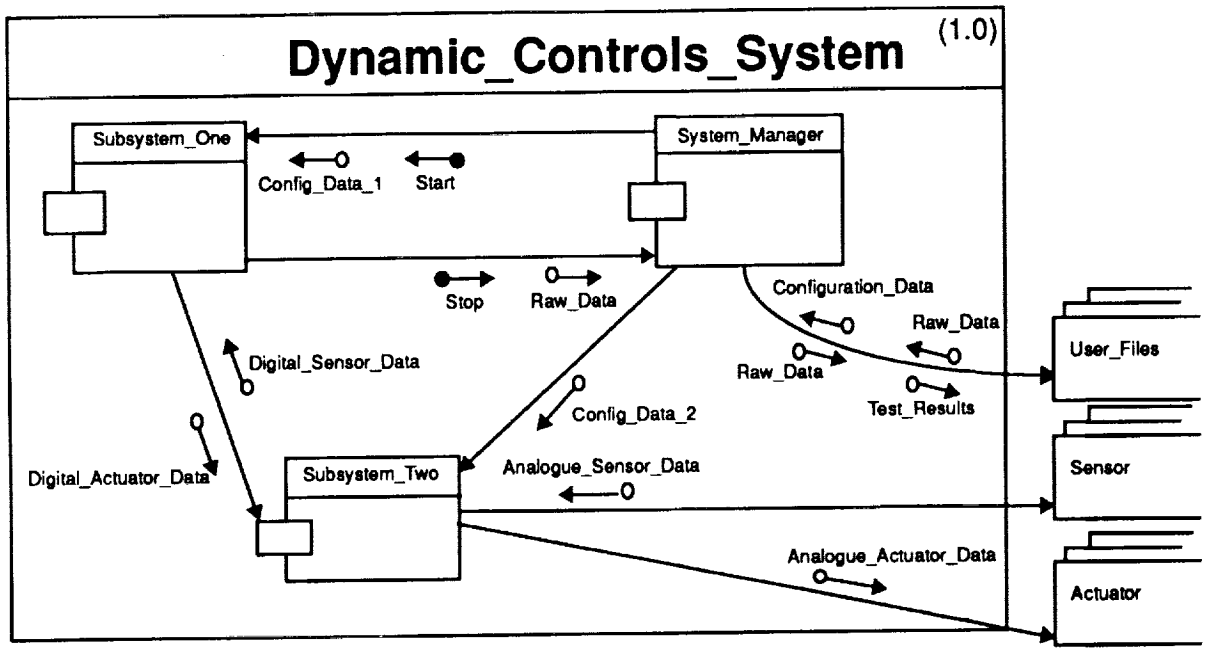


Figure 4. The parent object Dynamic_Controls_System is decomposed into 3 child objects: System_Manager, Subsystem_One, and Subsystem_Two. In the delivered system, all communications between the 3 objects were conducted over a MIL-STD bus.

Object control and communications were defined by diagraming the control arrows and data flows. All the data flows were logged in a data dictionary. A definition was written for each data entry and any applicable requirements were also referenced. Notice that there are control arrows pointing in both directions between Subsystem_One and System_Manager. When the system is being configured, System_Manager is in control. After configuration, Subsystem_One assumes control of both of the other objects. This type of information, which is not recorded on the diagrams, is logged in the Object Descriptions for each of the objects involved. For example, state transition can be recorded in a state transition table and references to that table can be included in the Object Descriptions for each of the objects affected. The command which causes a state to change can be diagramed using data couples as shown by the Start and Stop commands in Figure 4.

Once the requirements, control arrows, and data flows had been specified it was possible to identify the operations on the child objects. For each operation identified, an Operation Description was drafted (Figure 3). A

brief description of what the operation was required to perform was recorded. The object's requirements were then assigned to specific operations and those assignments were logged in the appropriate Operation Description form. It is important to note that, all the operations on the objects and all the inputs and outputs to the objects had been thoroughly documented both graphically and textually with the use of the object diagrams and the description forms. Therefore, each object had a clearly defined interface. By first assigning all the system requirements to the three child objects, and then thoroughly defining the interfaces between those child objects, the complexity of the remaining design decomposition was considerably reduced. It was then possible to concentrate on the decomposition of a particular child object, and its requirements, to the exclusion of all others.

System_Manager Object Decomposition

The first level of decomposition was very straightforward since there was a one-to-one correspondence between the computer subsystems and the first level of child objects. However, the decomposition of the System_Manager was not as straightforward. Far too many objects had been identified for a single layer of decomposition and there was no apparent way of grouping them into a logical hierarchy. (A goal of seven, plus or minus two, objects per level of decomposition had been established to minimize the complexity of the design.) The System_Manager had three states of operation; configure the system for a test, record raw data during the test, and post process the raw data. To reduce the complexity of System_Manager it was decomposed into three state manager objects; Pre_Test_Manager, Test_Manager and Post_Test_Manager (Figure 5). Part of System_Manager's requirements were delegated to the internal operation Execute, which scheduled state transitions by making the appropriate calls on the state manager objects. After all the operations had been defined and documented, the remaining requirements for the System_Manager were then decomposed and allocated to the three child objects. For each, an Object Description was written in which the requirements allocations were recorded. All control arrows and data flows were then diagramed and the data dictionary was updated. All operations on the child objects were identified and their Operation Descriptions were completed. These graphical and textural descriptions thoroughly defined each object's interface. It is important to restate that, the number of objects required to define System_Manager were reduced by breaking the requirements into logical groupings (by state) and using state manager objects to encapsulate those groupings. As a result, the design was partitioned in a way that made it possible to concentrate on the decomposition of a particular state manager object, to the exclusion of the others.

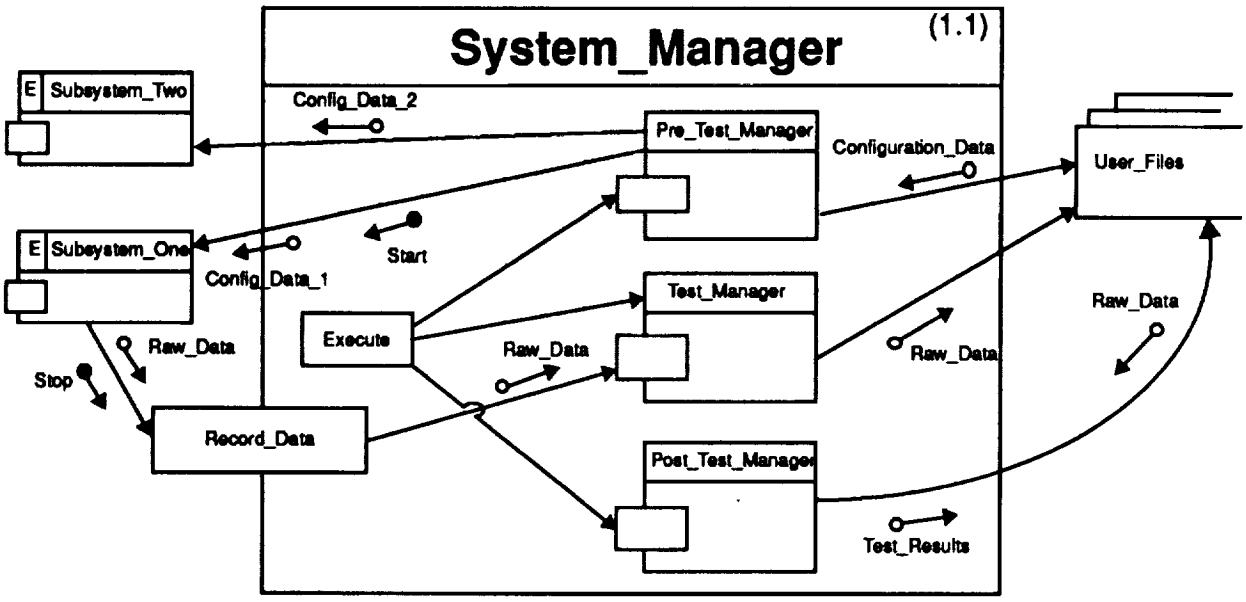


Figure 5. System_Manager is decomposed into 3 child objects: Pre_Test_Manager, Test_Manager, and Post_Test_Manager. System_Manager has one internal operation, Execute.

Pre_Test_Manager Object Decomposition

Pre_Test_Manager was the first state manager to be decomposed. Stepping through the requirements, in the order in which they would be performed, revealed which objects would reside on this level of the design. The first executable requirement of the Pre_Test_Manager was to obtain data for configuring the system. The configuration data was kept on three user-supplied files. These were the THINGS that were needed to fulfill the requirements. Therefore, a child object was created for each of those files; Script_File, Control_File and System_File (Figure 6). These file objects would provide, to Pre_Test_Manager, operations for obtaining the required information. In this way the details of how the configuration information was obtained and file manipulation achieved was hidden from Pre_Test_Manager by the three file objects. Therefore, Pre_Test_Manager could simply make a call on the file objects to satisfy the requirement (obtain data for configuring the system).

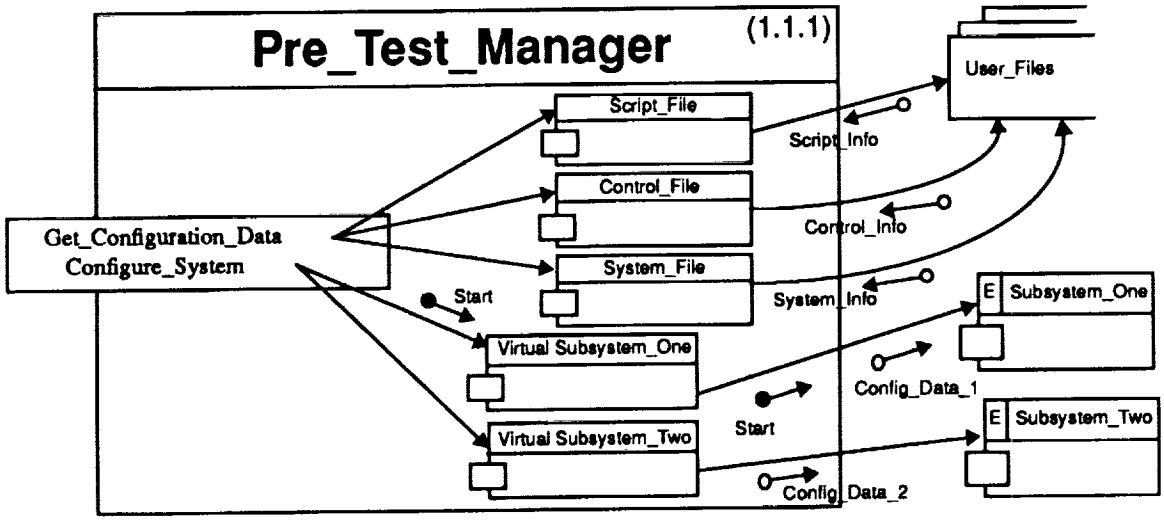


Figure 6. The Pre_Test_Manager is decomposed into five child objects, two of which are virtual objects. Note that the 'E' in the upper left corner of an object designates it as an external object.

Pre_Test_Manager's second requirement was to configure the subsystems with the user-supplied data. However, this was a distributed system and all communications between the System_Manager, Subsystem_One and Subsystem_Two were transmitted through a MIL-STD bus. A bus object was needed to communicate to the other two subsystems. But it was inappropriate to include a bus object at this level of the design, since a strong coupling between bus-related objects and application-related objects at this level of decomposition would substantially reduce the portability and reusability of the resulting components. Therefore, a virtual object[2] was created for both subsystems (Figure 6) [2]. Virtual_Subsystem_One and Virtual_Subsystem_Two would provide, to Pre_Test_Manager, operations to configure the system. Therefore, the complexity of Pre_Test_Manager's decomposition was further simplified by using virtual objects which encapsulated the details of bus communications.

Script_File Object Decomposition

For this case study, assume Script_File had only one operation, Obtain_Script_Data (Figure 7), and all of Script_File's requirements were allocated to that operation. Stepping through those requirements in the order in which they would be executed revealed that opening a file would be the first requirement executed. Therefore a child object, File_Manager, was created. The File_Manager was allocated the requirements for opening the files and handling errors which occurred in that process. As execution continued information would be taken off the file and

2. A virtual object is a logical construct used to represent an external object that resides on a separate processor. The virtual object imitates the external object's interface. An external object is an object which is allocated to a separate subsystem.

put in storage for later use in configuring the system. To do this the child objects, Sensor and Actuator, were created to store information relating to the system sensors and actuators.

Collectively, the three file objects; Script_File, Control_File and System_File provided a partition between the dynamic controls domain and the file management domain. That resulted in a decoupling of the domains. Therefore, the system was more maintainable since changes to the controls domain would not affect file objects and changes to the file system would only effect the file objects and their encapsulated child objects. For example, if a requirements change specified that the actual script file was to be obtained from a network node instead of a file on the disk, the File_Manager object could simply be replaced with a Network_Manager object. Since the Script_File encapsulates all the design elements used to implement input operations, Pre_Test_Manager would be unaffected. In addition, portability was increased since File_Manager was designed to provide general operations having to do with file access so that it could easily be reused. Not only was it reused by the Control_File, System_File and objects in Test_Manager and Post_Test_Manager but it could be reused by other systems in other domains which require disk file access.

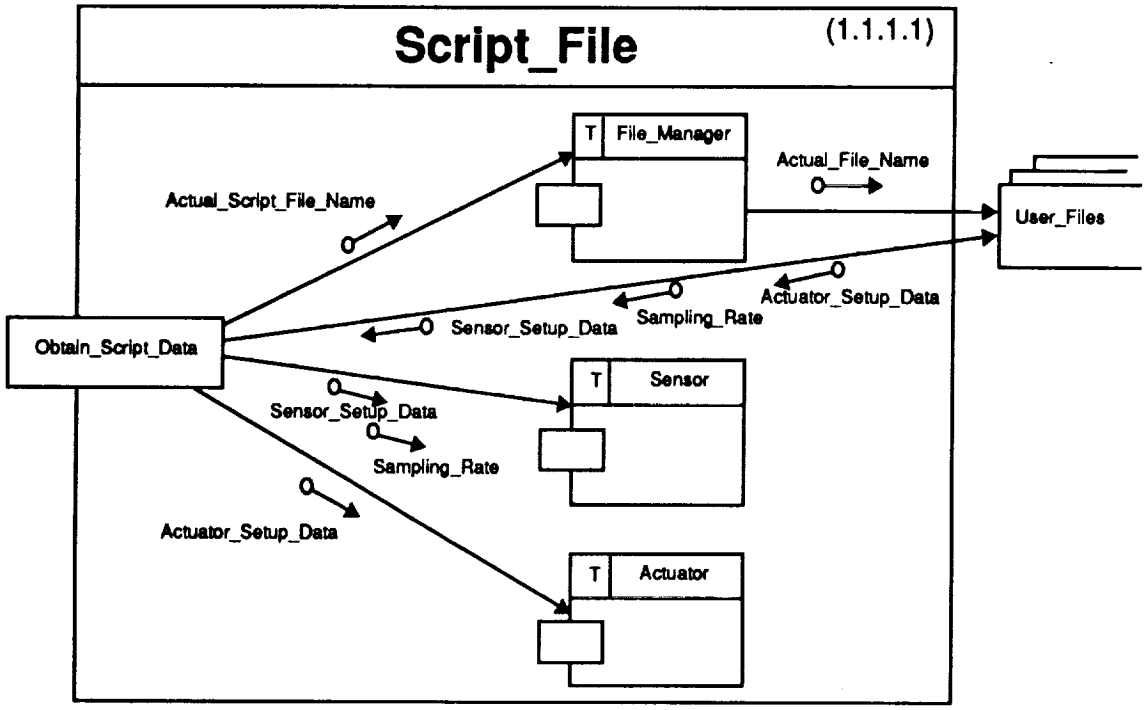


Figure 7. Script_File is decomposed into 3 child objects: File_Manager, Sensor and Actuator. The 'T' in the upper left-hand corner of the child objects indicates that they are 'Terminal Objects'; objects that can not be decomposed any further since all requirements allocated to them are executed by one of their operations.

Virtual_Subsystem_One Object Decomposition

To perform the operation Configure, Virtual_Subsystem_One needed to access the Sensor and Actuator objects to obtain the information necessary for configuration (Figure 8). That information had been placed in the Sensor and Actuator objects by the three file objects; Script_File, Control_File and System_File. To transmit that information to Subsystem_One, a Bus_Manager was created to encapsulate the details of the communications domain. Since bus management would require complex hardware specific code, it was decided that two separate design efforts would be conducted in parallel: first, the application-level design which dealt with the real world dynamic controls domain; and second, the design of the communications drivers for the MIL-STD bus. The communications driver design was done bottom up, from the card level. Together, figures 8 and 9 graphically show how the two designs were merged. The top level object from the bus design was Bus_Manager. It provided, for

example, 'get' and 'put' operations to Virtual_Subsystem_One. In the same manner Virtual_Subsystem_Two reused the Bus_Manager to communicate with Subsystem_Two.

Portability was substantially increased by creating a hierarchical design in which virtual objects were used to partition the application domain components from the bus domain components. For example, controls domain components could be ported to other systems having different communications devices. In addition, the bus communication components could be used to control bus traffic for any application using the same MIL-STD bus and card. Over four thousand lines of code from Bus_Manager have already been reused on another project, and no modifications were necessary even though the application domain was completely different. This was possible since Bus_Manager provided general purpose operations to implement the MIL-STD bus protocol which had no relation to the application domain.

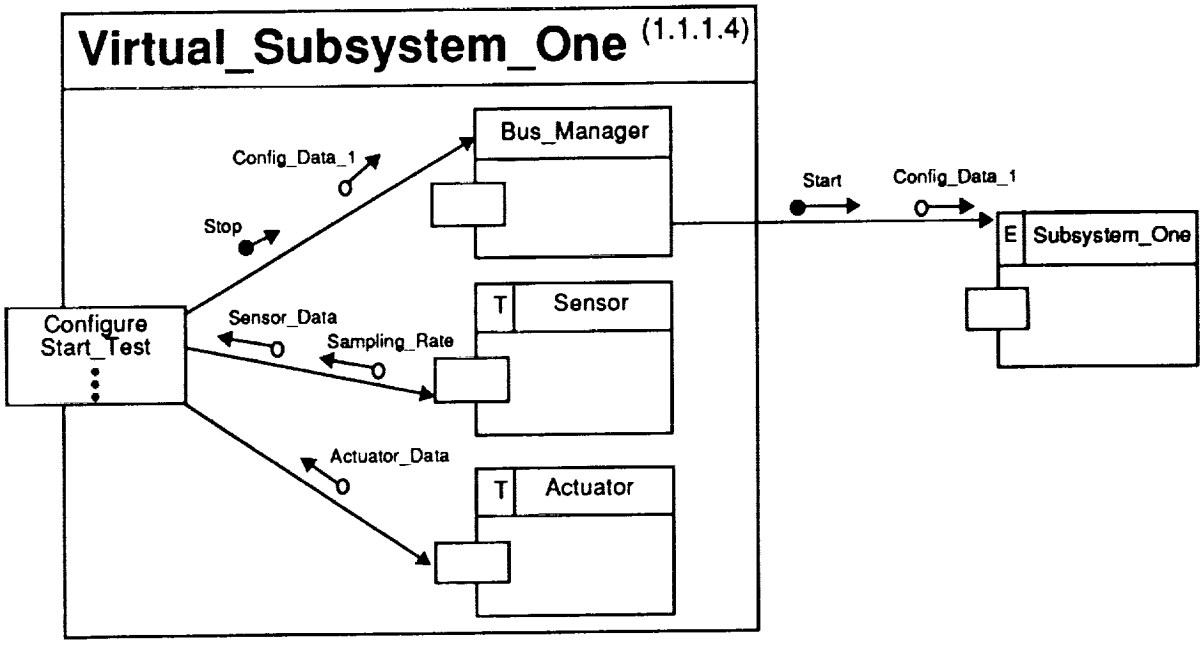


Figure 8. The Virtual_Subsystem_One is decomposed into 3 child objects: Bus_IO, Sensor, and Actuator. Note that Bus_Manager (figure 6) and its child objects facilitate access to the external object Subsystem_One.

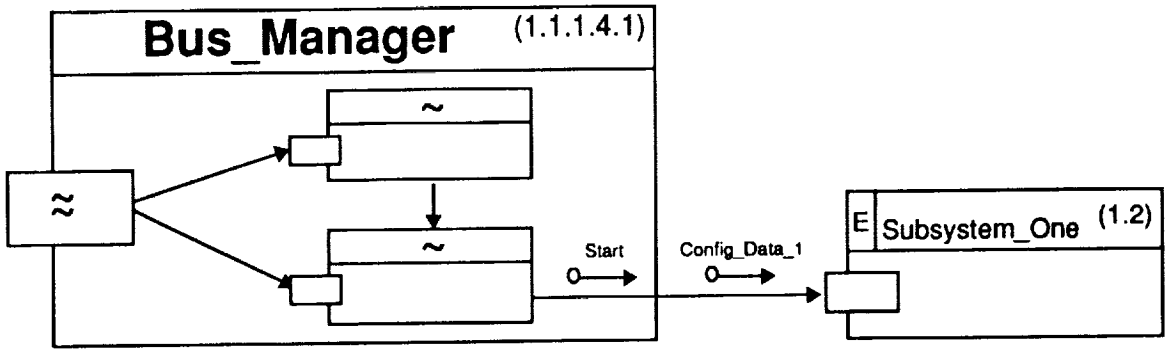


Figure 9. Bus_Manager is decomposed into two child objects which contain routines that control communications across the MIL-STD bus.

Extensions to the Preliminary Design Decomposition Steps

After reviewing the completed preliminary design it was evident that certain generalizations could be made about the decomposition process which could augment the preliminary design decomposition steps shown in Figure 2. These were recorded to provide additional insight into the design process for future projects and to confirm that these concepts worked successfully.

Virtual Objects.

If the THING that is needed to fulfill a requirement is an external object residing on a separate processor:

(A) Create a virtual object to represent that THING and assign to it the operations required (by the parent) to manipulate the external object.
(B) Then create a child object to manage the details involved in controlling the communications device used to access the external object. The communications manager object should provide only those operations specific to the defined protocol for that device.

Support Domain Access.

If services from a support domain, such as file management, are required to access the THING that is needed to fulfill the requirements:

(A) Create an object that will represent that THING and assign it the operations necessary to fulfill the parent requirements.
(B) Create a child object to manage the implementation of the services required by that support domain. This domain manager object should provide only those operations required to manipulate elements under its domain.

Both of these techniques are used to partition the design so that objects related to different aspects of the solution are loosely coupled which increases the portability of the resulting software components. Also, the domain/device manager objects encapsulate implementation details and provide a controlled interface through which services are obtained. This increased the maintainability of the resulting system in two ways: first, any changes related to the domain/device would be localized to the encapsulating object; and second, modifications to other objects would not effect the internal implementation of the domain/device object.

State Managers

If the parent object has several states, and a number of objects associated with each state, a child state manager object should be created for each of the states to reduce the complexity of the remaining design decomposition.

Mixing activities from preliminary and detailed design is one of the most common mistakes designers make. It is important to refrain from considering implementation details or data types until the detailed design phase. During the preliminary design, emphasis should be placed on what objects are necessary to fulfill the requirements, rather than on how requirements could be implemented.

DETAILED DESIGN

The general rules for transitioning from preliminary to detailed design were fairly straightforward. All the objects and operations were converted to Ada Program Design Language (PDL). Each object was made into an Ada package or task. Each operation was made into an Ada function or procedure and the data flows and the data dictionary were used to determine the data types for the operation parameters. Any alterations, additions or deletions in the design were documented by updating the preliminary design documentation. The Object and Operation Description forms from the preliminary design were reused to document the detailed design. The descriptions were

copied into the prologues of the packages and operations. Since these descriptions documented the requirements allocated to each preliminary design element, traceability from requirements to detailed design was maintained. Also, the algorithms from the Operation Descriptions were inserted, along with null statements, into the Ada functions and procedures. The design elements were then compiled to verify the Ada interfaces. In addition, the PDL and the code were both done in Ada, so the process of converting the PDL to the completed code was just a matter of coding the algorithms specified within the PDL. Since the code also contained the documentation which specified the allocated requirements, traceability from the requirements to code was also achieved.

DESIGN DOCUMENTATION

A well defined method of documentation is invaluable. It basically eliminated the subjective nature of the preliminary design diagrams. The Object and Operation Description forms (Figure 3) supplement the object diagrams and provided an opportunity for the designers' intentions to be documented. Although it has not been discussed in this paper, library numbers were used to uniquely identify each graphical element of the design. To assure that the proper description was associated with each graphical element, those numbers were also recorded in the description forms [3]. When the preliminary design was completed the diagrams and accompanying description forms contained enough information to implement the detailed design. In addition, the description forms were used to trace the requirements allocation and build the requirements traceability matrix. A Decomposition Tree was also made which pictorially represented the parent/child hierarchy [3]. This was used as a quick reference guide and also as an aid in locating reuse opportunities along different branches of the design. In addition, it can also be used by management to track the progress of the design activity. An accurate representation of the current projects configuration can be maintained by updating these documents during each phase; detailed design, implementation, testing, and delivery. Collectively these documents can serve as the 'As Built Configuration Document' which describes how the functional specifications were achieved in the final product.

PROCESS IMPROVEMENT

Many strides were made in OOD process improvement during this project. The most significant of these was to clearly define the process itself. Figure 10 shows a graphical representation of the process. By determining the process, as well as the steps and procedures followed at each phase, a baseline for process improvement is defined. As future projects reuse this process, procedural improvements can be added to the baseline and the list of lessons learned can be augmented. This information can also be exchanged with other organizations using similar methods. To facilitate this, an individual in each organization is given the responsibility of recording the current state of the process, discovered improvements, and lessons learned. Not only are improvements and lessons learned recorded, but an attempt is made at documenting the rationale behind them. Each organization is responsible for feeding this information back to a central person, the 'keeper of the method.' This person is responsible for collecting from each organization the improvements and rationale, updating the method accordingly, and redistributing it to all the organizations involved. So far, these organizations include two NASA centers, ESA, and several commercial companies. Although this network is in its infancy, it is spreading nationally as well as internationally.

CONCLUSION

With the OOD procedures outlined in this paper, the complexity of the preliminary and detailed design process can be substantially decreased. In addition, the reusability, portability, and maintainability of the resulting products will be increased. Also, process improvement can be obtained by documenting the details of the procedures involved and incorporating successfully demonstrated improvements into those procedures.

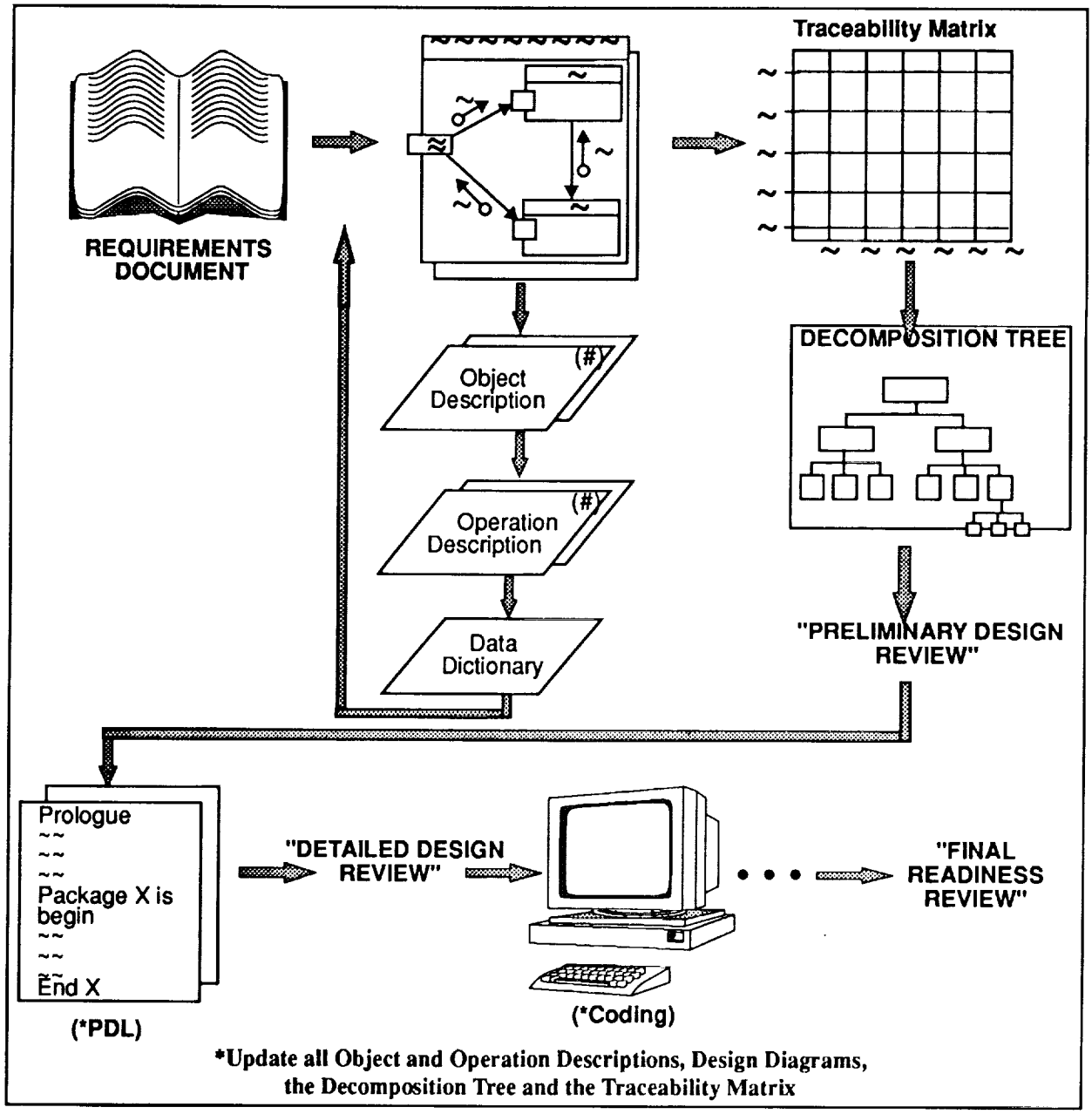


Figure 10. Process diagram.

REFERENCES

1. [Anderson 91] Anderson,J.,et al., Manageable Object-Oriented Development: Abstraction, Decomposition, and Modeling, Proceedings of Tri-Ada'91, San Jose, CA., October 21-25, 1991.

2. [Mc Quown 89] McQuown,K.L. Object Oriented Design In A Real-Time Multiprocessor Environment, Proceedings of Tri-Ada '89, Pittsburgh, PA., October 23-26, 1989, pp. 570-588.

3. [Schuler 91] Schuler,M.P.,Evolving Object Oriented Design, a Case Study, Proceedings of the Eighth Washington Ada Symposium (McLean, VA., June 17-21, 1991), pp.50-61.