ORGANIZATIONAL BARRIERS TO OBJECT-ORIENTED DEVELOPMENT

John Glaize, Staff Scientist
Link Flight Simulation Division of CAE-Link Corporation
Binghamton, NY

ABSTRACT

The Defense Department has instituted an aggressive initiative to cut software costs by requiring contractors to build reliable, maintainable, and reusable software. The STARS program and the Ada programming language are an outgrowth of this initiative. An effective software development methodology for meeting these requirements is Object-Oriented Development (OOD). However, experience has shown that the adoption of OOD is not only a technical issue but an organizational issue as well. There can be significant barriers to successful Object-Oriented Development if the engineering organization is structured along functional lines, as many companies currently are. Examples of these barriers are in the areas of software ownership and management, cost accounting and work breakdown structure, and geographical considerations. In order to overcome these barriers and to exploit the full benefits of OOD, a company may need to analyze its structure and be open to potential changes to its culture, its engineering policies, and its software organization.

INTRODUCTION

Realizing that the incorporation of new software development methodologies is not always an exact science, CAE-Link has afforded its engineering staff the entrepreneurial freedom to explore alternate, possibly radical, approaches for achieving this incorporation. This paper resulted from such an exploration performed in connection with the software development on the B-2 Aircrew Training Device (ATD) Project.

One of the most significant emerging requirements for ATDs is device availability. The users of training devices have become aware through painful experience that an ATD has very little utility unless it is up and operational for a high percentage of the time for which it is needed to provide training. To support this need, the specifications for current ATDs often require the device to have availability rates of 95% or better. This means that after the ATD has been deployed, the time used for isolating and repairing problems or for making modifications can be no more than 5% of the potential training time!

This requirement imposes severe constraints on the developers and maintainers of ATDs. Why is availability so critical? First, the effects are of long duration. The benefits of high availability or the detriments of low availability are felt throughout the entire lifetime of the device. Secondly, the complexity of ATDs has increased dramatically. This increases the need for the capability to isolate and repair problems quickly because the probability of latent problems is higher. In addition, the probability for potential modifications has increased. Rapid changes in technology and development of new capabilities result in frequent changes to the aircraft, and to be useful, the ATD must keep in step. Lastly, many ATDs are being developed concurrently with the actual aircraft, which means that aircraft data, characteristics, and on-board equipment are constantly evolving. The added complexity and rapidly changing nature of the aircraft have underscored the need for operational trainers while, at the same time, making it increasingly more difficult to achieve this availability.

Even though the availability requirement pertains to the maintenance phase of the ATD, it can only be accomplished if availability considerations have been built into the basic design of the device. In order for the device to achieve the required availability ratio, the designer/developer must employ a software design methodology that supports maintainability (which is defined to be the extent to which errors can be isolated and corrected) and modifiability (which is defined to be the extent to which modifications can be made effectively). Object-Oriented Development (OOD) has been recognized as a methodology that has advantages over Functional Decomposition for meeting these requirements. Many companies, with the enthusiastic consent and encouragement of their customers, have based their software architecture on an object-oriented paradigm in a conscious attempt to enhance ATD availability.

However, the engineering departments of these companies are often structured along functional disciplines. The major thesis of this paper is that there may be significant barriers to effective OOD when attempted within the context of a functionally oriented engineering organization. In other words, the successful implementation of the OOD development methodology is a management issue as well as a technical issue. In fact, the advantages of OOD may be limited if the company views its incorporation as only a technical issue. This paper describes some of the organizational barriers that may exist, and proposes ways of eliminating or mitigating their effects.

WHAT IS OOD?

Before describing these barriers, a brief discussion is presented which outlines the principles of the OOD methodology, some of its advantages for enhancing ATD availability, and its relationship to the Defense Department's standard programming language, Ada.

OOD is a software development methodology based on constructing a system from a collection of software modules called objects. "An object comprises both a definition of a data [structure] and the operations that may be performed on that data." [2]. Therefore, "systems are modularized on

the basis of their data structures" [1], rather than on the basis of functionality. Moreover, all of the operations on that data structure, i.e., all processing routines that affect that data or have knowledge of its form, are contained within the same software module as the data itself. For an ATD, examples of objects are fuel tanks, radios, targets, wind gusts, or aircraft position and attitude.

The advantages of OOD for ATD development are manifold. "A statement of what a program does is far less useful than a statement of what the program is." [6] The software is easier to understand because the software modules (the "solution space") more closely resemble the real world entities (the "problem space"). "Systems more closely model the real world, and are easier to maintain and modify" [3]. Due to the closer association of the problem space and solution space, the effects of changes tend to be better isolated; a change to an entity in the real world affects fewer software modules. In addition, it is generally easier to identify the modules that are affected. The OOD methodology realizes these advantages especially well in data-driven applications, where the modifications to be made are manifested in changed data specifications. An ATD is largely a data-driven application, and thus is well suited to this methodology.

In addition, there exist two basic principles of software engineering that help manage the complexity of the development of large systems: abstraction and information hiding. Abstraction involves the identification of the properties that are essential to the problem, while omitting unessential details. Information hiding involves encapsulating in one place those details which do not affect the rest of the system, and making them inaccessible to the rest of the system. Judicious use of these two principles tends to produce higher-quality software. Object Oriented Development, by abstracting the essential details of objects and encapsulating their data representation with their operations, supports both abstraction and information hiding. "Abstraction and information hiding form the foundation of all object-oriented development. [4]"

Object Oriented Development also bears a natural relationship to one of the Defense Department's emerging requirements: the use of the Ada programming language. Ada was designed to embody software engineering principles such as abstraction and information hiding by providing such programming structures as packages. According to Donald Reifer, a noted consultant in the Ada community, "Ada is primarily a packaging technology." An object, which consists of the definition of a data structure along with the corresponding operations, is generally implemented as an Ada package (the implementation of an object is often called an Abstract Data Type or Abstract State Machine). Indeed, Ada has been classified as an "object-based" language because of its support of abstraction and information hiding. Ada and OOD form a natural partnership for meeting the software engineering objectives of maintainability and modifiability, and thereby promoting ATD availability.

For these reasons, OOD is a methodology worth incorporating!

A REPRESENTATIVE OBJECT

But, as stated above, organizational barriers to effective OOD can exist. As a means for illustrating these organizational barriers, a typical object, a fuel tank, is now defined. Within the single software module (package) that defines the fuel tank are contained the data that represent the abstraction of the fuel tank (i.e., the attributes of the tank that are pertinent to the application) along with all operations that affect the tank.

Examples of pertinent fuel tank attributes for an ATD may be such entities as capacity, number of compartments, current quantity of fuel, current temperature of fuel, number of valves, etc. Other attributes of a fuel tank, such as the type of metal from which it is made, are not pertinent to an ATD and therefore are not included in this abstraction. Examples of operations on the tank may be a function for computing current quantity, a function for determining the effect of the quantity on the weight and balance of the aircraft, a function for preparing the quantity for display on an instructor station, etc.

Therefore, many different types of operations may coexist within a single OOD software module. The operations are grouped together because they all pertain to the object (i.e., the fuel tank) and require knowledge of its structure, not because they perform similar functions. A properly constructed object is internally cohesive (all operations within the package are related to the object) and loosely coupled to other objects (all operations that pertain to that object are part of the package). In this way, the effect of any changes to the real world fuel tank tend to be isolated to the fuel tank object.

BARRIERS TO OOD

Even though OOD is an advantageous software development methodology to adopt for ATD production, there can exist significant organizational barriers to be considered when the engineering departments are structured along functional lines. Some of these barriers are Software Ownership, Documentation Ownership, Work Breakdown Structure and Time Charging, Management, and Logistical Barriers.

Software Ownership refers to the organizational and cultural need for departments and individual engineers to "own" software. For each software module, a section of the organization must be assigned that is responsible for the module's design, development, and control. Culturally, engineers are often very protective of their software. In a functionally oriented software development methodology, this assignment is generally straightforward and maps logically to the functional organization.

However, when using an OOD methodology, the organization must encourage several functional groups to contribute functions to a single software module. This is because the unit of modularity, an object, is operated upon by several different types of operations. In the example of the fuel tank presented above, the function for computing tank quantity may be provided by a fuel flow expert, the function for weight and balance by a forces expert, and the function for display by an instructional systems expert. In a functional organiza-

tion, these individuals are generally in separate departments. Which department owns the fuel tank object?

The answer is that these objects are not the sole property of any one department, but belong to the system as a whole. Such a concept, however, is usually anathematic to a functional organization; this type of software ownership appears to be too loosely defined and unmanageable. In addition, some functional groups begin to develop an identity crisis. Their software doesn't "belong" to them any more, but instead is scattered and integrated into "someone else's" software. In a culture where software ownership is closely related to department affiliation, a group can feel that their identity is threatened by OOD.

The second barrier, Documentation Ownership, is akin to Software Ownership, but manifests itself in the documentation. This issue is described separately because the production of high quality, useful, specification-compliant documentation is so often such a difficult process that adding additional complexity is especially onerous to management.

If the software is "centrally owned", should the documentation also be owned by the system? For example, should the design of the fuel tank be contained in a single design document whose component parts are collected from several different functional departments, or should each department maintain its own documentation that contains only those parts of the software for which that department is directly responsible? If there is one central document, how is it managed so that all component parts are consistent and up-to-date? There may even be problems in getting such an eclectic document to "read correctly". Production of object-oriented documentation is difficult for a functional organization.

Work Breakdown Structure and Time Charging barriers are even more significant than Ownership issues because they involve not only engineering but also accounting and possibly legal considerations as well. Companies must pay scrupulous attention to proper time accounting or possibly face dire consequences, especially on Government contracts. Shoddy practices in this area can result in withholding of payments, heavy fines, or loss of bidding privileges. Convictions can lead to severe loss of profit, or even put the company out of business! Companies must ensure that the work allocation and the work being performed correlates correctly with the time accounting.

This correlation usually falls out quite logically when the organization and software products are structured along similar lines, such as when both are functionally oriented. The engineers in a particular functional department charge to that department's accounts while developing "that department's" software. When the software structure is object-oriented and the organization is functionally oriented, however, the time charging may not be quite so clear. For example, when an engineer assigned to the Instructional Systems department develops a display function for the fuel tank object, is that engineer working directly for the Instructional Systems department (and therefore should be charging an Instructional Systems account), or is the engineer "subcontracting" to the Aircraft Systems department (and therefore should be charging an Aircraft Systems account), or is the engineer merely working on the fuel system (and therefore should be charging a Fuel System account)? Any of the above answers may be correct depending on the structure of the accounting system. The point, however, is that the way in which the company's accounting system and work breakdown structure is organized may need to be scrutinized and possibly revamped so as to record time charges correctly and legally, and provide management with relevant and useful status information.

This leads to the next barrier: Management. As in earlier barriers, since there is an "apparent disconnect" between the structure of the organization and the structure of the software products, the functional managers may have a more difficult time performing their management duties. The traditional management metrics for monitoring employee's performance and statusing the progress of the software are not as straightforward to apply because the software products are not as closely associated with the department and with individual engineers. Whereas traditionally a manager could assign a functional software entity to an engineer and then status that entity, the engineer's work now tends to be scattered among several software entities, and each entity represents the work of several engineers. This requires new techniques of "dipsticking" in order to measure progress and earned value.

A related concern is the management of computer resources. In order to manage the use of computer time and memory, and to provide extra quantities for spare and growth potential, a department is generally given a resource budget. With a functional development methodology, the software belonging to a particular department is typically allocated to a particular computer processor or set of processors. This facilitates the comparison of resources expended to resources budgeted. With an object-oriented software structure, the resource budgets are assigned to objects. Since several departments may contribute to the development of an object, it is more difficult to manage this budget and to correct overruns. Who is responsible if the fuel tank object described above is over budget? How are recovery plans formulated?

Logistical barriers can exist when major functional divisions of the system are produced in different geographical regions of the company, or are subcontracted to other companies. With a functional organization, it is quite common and natural to "carve out" a major function, such as a visual subsystem or a mission generation station, and assign the development of that subsystem to a group of engineers that is physically removed from the other system developers. If a similar strategy is employed when an object-oriented development methodology is being used, it is more difficult to integrate the work of these engineers into the software objects that comprise the system.

A case could be made for keeping these major functional subsystems removed from the software objects. How could something like a visual subsystem exist as part of other ob-

jects, and why would such an attempt even be made? As foreign as this concept might first appear, it may actually be advantageous from a software engineering standpoint to embed visual generation functions within software objects. Thus an object that represents a target in the gaming area could possess a visual generation function to produce the visual image of the target, similar to the display function for the fuel quantity that was discussed in our representative fuel tank object. Even though this concept appears extremely radical, it may actually promote maintainability and modifiability by isolating the effects of any latent errors that exist in the target simulation, or by easing the incorporation of new or different kinds of targets.

The fact that companies that are organized along functional lines have traditionally identified major functions and physically removed their developers may pose logistical barriers to the production of highly cohesive and loosely coupled objects. The software developed in other geographical locations tends to be structured functionally instead of incorporated into the ATD system objects. This section of the paper has described several barriers that exist when attempting an OOD development methodology within a functionally oriented engineering organization. Although these barriers manifest themselves in different ways, their genesis derives from a common fact: the software development methodology and the underlying organization must be aligned so as not to function at cross purposes.

ALTERNATIVE SOLUTIONS

There are several ways by which a company may choose to deal with the dichotomy of a functionally oriented organization and an object-oriented development methodology. Three alternative solutions that are discussed here are: Retain the Functional Organization With Little or No Change, Implement an Object Oriented Organization, and Retain the Functional Departments but Implement Object Orientation at the System Level.

The first solution, Retain the Functional Organization With Little or No Change, represents the easiest to implement from a management point of view. The current organization remains intact, the current managers continue to manage functional groups, and engineers are not reassigned. The engineers are merely instructed to develop their subsystems with an object orientation.

Even though this approach has the least impact on the organization, experience has shown that it also provides the least potential benefit. The goals of OOD are often realized only in a limited fashion, because the development of maintainable, modifiable, and reusable objects occurs only at very low levels, i.e., where all of the applicable operations associated with the object fall within the natural purview of an existing functional department. With a functional organization it may be difficult to extend the advantages of OOD to the system as a whole.

There are several reasons why the advantages are not fully realized. First, even though the traditional set of operations on an object or a subsystem is generally believed to contain "all applicable operations", it usually contains only those operations that are of interest to a particular functional department. For example, when a fuel tank object is developed by an aircraft systems department, it typically contains only operations that pertain to the fluid dynamics of the fuel tank. Other operations, such as functions to display attributes of the fuel tank on an instructor station, are still physically implemented in a different software module (such as a "display object") that is developed by a different functional department. The major reason for splitting these operations into two different objects is the functional orientation of the organization.

The result of this division, however, is that it is necessary to pass information about the fuel tank from the fuel tank object to the display object. This leads to objects that are more tightly coupled and increases the number of interfaces. When problems exist, it is more difficult to identify the location of the problems and has a negative effect on maintainability.

A second reason for diminished benefit is that information hiding usually is not exploited. A powerful feature of Ada for providing information hiding is the private type. In Ada, "objects are denoted by instances of private or limited private types" [5]. Implementing an object as a private type allows all information about the object that is not needed by other software to be "hidden" within the object package. This hiding tends to localize the effect of changes and thereby enhance modifiability. It follows that not exploiting information hiding has a negative effect on modifiability.

A functional organization usually finds it difficult to make widespread use of private types. Often this stems from a traditional division of labor that implements certain "system-wide" functions as separate software instead of integrating these functions within the objects. This includes such functions as management of various ATD modes (e.g., real-time or mission freeze), saving and restoring the current state of the system for repeating portions of the mission, or debug functions used for checking out the software. Since these functions are implemented separately, they need to know the structure of each object in order to save its state or examine it for debugging purposes. This need for a global view precludes the implementation of the objects as private types.

An example from the author's experience illustrates the detrimental effects that can result from not hiding information. A particular software system that performed configuration management contained an object that represented software part numbers. The maximum size of this object was set at eight characters (i.e., a part number could consist of no more than eight characters). This piece of information was not hidden from the other software in the system, however, with the result that many software modules, including those that created operator menus or generated reports, were predicated on the eight-character maximum. When the requirements changed, and it became necessary for the part number to be greater than eight characters, the effect of the change was felt throughout the system, many modules had to be redesigned, and a significant period of time was needed

to incorporate the change. If information hiding had been used, the other portions of the system would have been required to query the part-number object to determine its size, and the change would have been isolated to that one object.

Therefore, even though the impact on the organization is minimized, a company that is planning to adopt OOD as a development methodology without contemplating any corresponding organizational changes should be aware that the theoretical benefits of OOD will be diminished significantly. Forward-looking companies should investigate the possibility of moving beyond this alternative.

The second solution, Implement an Object-Oriented Organization, represents the other end of the solution spectrum. In this case, the potential benefits of OOD will likely be maximized, but the impact on the organization will be far-reaching as well. In order to put this solution in place, general classes of objects are identified, and departments are formed that are responsible for the production of these objects. Typical classes of objects could be "engines", "electrical systems", or maybe even smaller entities such as "pumps". Note that the pump department, should one exist, is then responsible for all operations related to pumps, including computing pressures and fluid flows through the pumps, as well as preparing information about the pumps for display purposes.

The result of this effort is the production of robust, self-contained, reusable software object components, implemented as private types, that are placed "on the shelf" for use in other applications. Conceivably, a set of software pumps applicable to various ATDs could become as standardized as the set of hardware pumps used on the actual aircraft. As is probably obvious, however, the effects on the company of implementing this solution are quite drastic. The reorganization is massive, virtually all of the engineers are uprooted and possibly isolated from their current colleagues, and managers need to be trained in entirely different management techniques. In fact, the management methodology for this style of organization is substantially different than for a functional organization and would first have to be formulated. Such a change requires careful planning, and has to be phased in gradually to avoid total disruption. Due to the risk of unknown contingencies, it should be prototyped on a small project or as a research and development effort.

Another stumbling block to the implementation of this solution is that experience with the OOD methodology in the ATD application is relatively recent, and there is not yet general agreement on what constitute the "good" object classes for an ATD. Before the organization is so totally revamped, there should be a very high confidence factor, based on information gathered on several projects, that the object departments represent a proper and workable arrangement.

Therefore, a company contemplating an object-oriented organization should be aware of the degree of disruption inherent in this approach, and implement it carefully and gradually in a phased-in manner over a fairly long period of time. With proper implementation, however, this type of organization will probably realize the greatest advantage from the OOD methodology, and be able to produce very high quality software, while providing economic benefits to the company through enhanced reusability. Companies should seriously consider this alternative as a long-range goal.

The third solution, Retain the Functional Departments but Implement Object Orientation at the System Level, represents a compromise between the first two solutions. By the term "system level" is meant the level of the project that integrates together the various functional subsystems to form a cohesive training "system". The advantages of this solution are that better system-wide objects are produced than with the first solution, but the organization does not experience as disruptive a restructuring as with the second solution.

However, since this solution is reconciling object-oriented software with a functional organization, more management and coordination is needed at the overall system level. This coordination is achieved by the following organizational techniques: a Systems Engineering discipline to manage the overall system, a Software Engineering discipline to manage production of object-oriented software, a set of Functional Departments to produce the object operations, and a method of "internal subcontracting" to allow Systems/Software Engineering to produce specifications which are then subcontracted to the Functional Departments for implementation. These organizational entities are better defined by their responsibility and division of labor than by the type of personnel they contain. The Systems Engineering discipline is typically composed of senior engineers from the various Functional Departments along with overall Program Engineers who are concerned specifically with such issues as reliability and interface management. When the senior functional engineers are wearing their "Systems Engineering hats", however, they must be concerned with the overall ATD training system rather than particular applications.

The duties of Systems Engineering are to analyze the specification of the ATD and formulate a top-level system design. This design determines the overall functionality that the ATD must possess and outlines how each of the specification requirements will be met, as well as defining what functionality will be provided by hardware and what functionality will be provided by software. Systems Engineering designs the system architecture or "framework" that is used to connect all of the various software and hardware subsystems into a cohesive training device and supervises the actual construction of the device. Systems Engineering is then responsible for ensuring that all system components meet overall design criteria, and that they all function together properly to provide the required ATD functionality. This discipline is also responsible for developing the System Test Plan to coordinate the testing of the entire ATD, and for collecting and managing the overall ATD documentation. In this way, the device as a whole, is "owned" by Systems Engineering rather than by the Functional Departments. Although the device is owned by Systems Engineering, the actual development of the software components that are needed is "subcontracted" by Systems Engineering to Software Engineering.

Software Engineering is typically composed of lead engineers from the Functional Departments as well as computer scientists who understand and guide the object-oriented development process. The duties of Software Engineering are to analyze the software requirements allocated to them by Systems Engineering, and to produce a software design that outlines the overall software architecture for constructing the software subsystems and identifies the required software objects.

The software design developed by Software Engineering is at a lower level than the overall system design of Systems Engineering, but at a higher level than the detailed design of the individual object operations. Notice that in order to achieve the full benefit of OOD, these objects should be implemented as Abstract Data Types using the private type mechanism of the Ada language. The private type capability means that all operations that have knowledge of that object must be in the same package. Therefore, the design of the objects must specify the total functionality that the objects must possess. For example, the design of an object such as the representative "fuel tank" discussed earlier must consider not only "normal" functions such as fuel flow, but also such "special" functionality as saving the state of the object for trainer reset, and displaying the ACSII representation of the object on an Instructor Station terminal.

The Software Engineering discipline, after specifying the objects that are needed, determines whether each object must be built or whether it can be procured from the reusable object library. If it is decided that an object can be procured, an analysis is then performed to determine whether the object can be used as it currently exists or whether its functionality must be extended (i.e., whether additional operations must be developed). From this analysis, a total list of operations that must be developed is formulated. Software Engineering also develops the overall Software Test Plan used to test the entire software complement of the ATD, and coordinates the overall software documentation. In this way, the software objects, the software framework, the software documentation, and the software test plan are all "owned" by Software Engineering.

Software Engineering is responsible for the management and coordination required to pull the entire software system together. The actual development of the object operations is subcontracted to the Functional Departments by Software Engineering. Notice that operations for a single object may be subcontracted to several different Functional Departments. The duties of the Functional Departments are to analyze the requirements produced by Software Engineering, and develop the detailed design of the object operations. It is at this level that the actual algorithms and mathematical equations to accomplish the required operations are formulated. The Functional Departments produce individual documentation for each of the operations, which is then collected into coherent, specification-compliant software documentation by Software Engineering.

Finally, the Functional Departments design test procedures for the operations in accordance with the Software Test Plan developed by Software Engineering. Notice that the work performed by the Functional Departments in this organization is very similar to the work performed by Functional Departments in a strictly functional organization. The main difference is that the software (in particular, the structure of the objects) is owned and managed by an organization that transcends functional boundaries and limitations.

CONCLUSION

Seeking successful fulfillment of emerging requirements of Aircrew Training Devices has caused companies to re-evaluate the methodology by which they develop software. Many companies have recognized the benefits of object-oriented software in meeting these requirements. Perceptive companies realize that the benefits of these new development methodologies are maximized when an effective alignment exists between the methodology and the structure of the engineering organization. There are several alternative strategies for achieving this alignment; this paper has explored some of them. The innovative and forward-looking organizations will continue to investigate the effects of new methodologies, assess their position and structure, and make the necessary adjustments to sustain a technological and competitive edge.

REFERENCES

1. Meyer, Bertrand Object-Oriented Software Construction. Prentice Hall International Series in Computer Science, 1988

2. Petzold, Charles "Object-Oriented Programming." PC Magazine, January 17, 1989

3. Constantine, Larry Presentation at Object Oriented Symposium, Digital Consulting, Inc., Boston, MA, June 1989

4. Booch, Grady Software Engineering With Ada. Second Edition, The Benjamin/Cummings Publishing Company, Inc., 1987

5. Booch, Grady Software Components With Ada. The Benjamin/Cummings Publishing Company, Inc., 1987

6. Duntemann, Jeff "Structured Programming", Dr. Dobbs Journal, M&T Publishing, Inc., April 1990, pg. 137

ABOUT THE AUTHOR

John Glaize is a Staff Scientist and Ada Specialist for Link Flight Simulation Division of CAE-Link Corporation, where he has held various management and technical positions over the last 17 years. His primary task for the last four years has been to define a real-time Ada architecture and software development methodology to maximize the production of maintainable and reusable software. He holds a B.S. degree in mathematics from Carnegie-Mellon University and an M.B.A. from SUNY-Binghamton.