OBJECT-ORIENTED DEVELOPMENT OF TRAINING SYSTEMS USING ADA[R]

Dr. Matt Narotam
Burtek
Tulsa, Oklahoma

Dr. Sabina Saib
Thomson-CSF
Santa Barbara, California

Mr. Clifford Layton
Rogers State College
Claremore, Oklahoma

ABSTRACT

This paper discusses aspects of software engineering and design methodology to be used for development of software using Ada for an existing C-141B Operational Flight Trainer (OFT). The OFT software was originally developed using FORTRAN 77 as the implementation language. The paper describes the application of software engineering concepts such as abstraction, information hiding, modularization and generalization, and the development of a methodology for generating a program design based on these concepts. Also described are problems with the traditional object-oriented design (OOD) methodology, and attempts to front-end OOD with a functional decomposition technique for generating the system design. Changes are proposed to the OOD process to incorporate solutions for these problems. The methodology described is applicable to the development of software for any type of training device.

INTRODUCTION

The DOD mandate which specifies Ada as the programming language for future mission critical system will impact procurement of related training systems. Such procurement will be based on the results of current studies that can demonstrate the cost-effectiveness and feasibility of using Ada for software development.

In evaluating the impact of Ada on simulator engineering, consideration must be given to the benefits that will be gained from developing maintainable and reuseable software. Ada is a language rich in constructs that supports a design process oriented toward the development of such systems.

A development process that is based on FORTRAN restricts the application of concepts which bring about the development of maintainable and re-useable software systems. This is so primarily because of the primitive constructs available in FORTRAN and the lack of standardization which force designers to acquire a mind-set that encourages software design based on the peculiarities of various implementations of the language. This leads to the development of software systems which are costly to maintain and not easily reusable. Ada on the other hand supports a design process which encourages greater consideration to up front systems design requirements with appropriate prioritization of criteria such as maintainability, reusability, and system efficiency. This design process involves the use of a methodology to generate a software solution based on system requirements and system defintion. The process will lead to cost reductions as opposed to cost escalation. Crucial to this outcome is the application of the appropriate methodology for simulator systems engineering.

OOD is a methodology for developing Ada software utilizing software engineering principles. A complete presentation of the OOD methodology may be found in Booch (1) and EVB's OOD Handbook (2) and is referred to as traditional OOD in this paper. Traditional OOD employs the concept of producing a system solution using "real world" terminology which aids the process of maintainability and reusability of software systems. However, the steps identified in traditional OOD are not easily applicable to simulator software development. There are two major areas of weakness in implementing traditional OOD,

(1) Determining where the application of the methodology begins in a system engineering process, and

(2) The actual process of developing a system solution using real-world terminology.

The proponents of traditional OOD recommend the use of a front-end to OOD such as Structured Analysis Desgin Technique (SADT[TM]) (3) and Jackson's Systems Development (JSD) (4) for performing system analysis. Using a SADT based front-end, a functional decomposition of the system may be performed, at which point an OOD application may begin. This process is far from satisfactory for evolving the design for a simulator system.

[R] Ada is a register trademark of the U.S. Government (Ada Joint Program Office) Department of Defense.

[TM] SADT is a register trademark of SOFTECH, Inc.

Burtek, a subsidiary of THOMSON-CSF, is currently engaged in a research and development program for the USAF entitled the Ada Simulator Validation Program (ASVP). The ASVP is a proof-of-concept project intended to provide data to the Government for application on future procurements of training devices. The ASVP requires the redevelopment of software for an existing Burtek-built C-141B OFT using Ada.

This paper presents a refinement to traditional OOD. The Burtek refined OOD incorporates steps which expands OOD to perform *an Object Oriented decomposition of the system* followed by the implementation of a solution. The paper reviews traditional OOD and functional decomposition using SADT as a front-end to OOD. The problems with a SADT-based front-end and the traditional OOD process are then discussed followed by a presentation of solutions incorporated in the Burtek refined OOD.

REVIEW OF TRADITIONAL OBJECT-ORIENTED DESIGN

Object-Oriented Design (OOD) is a methodology that can aid creation and documentation of software design. The methodology furthers decomposition of a problem into "real-world" objects and manipulations (operations), rather than numeric data and/or functions. OOD should be applied to the result of the analysis and requirements phases of a software development project, and after detailed problem definition information has been collected; it should facilitate design which can then be translated into algorithms and code. OOD is intended to be supportive of client and designer communication, and Ada and software engineering concepts.

OOD has been developed with recently developed *object-oriented languages, such as* Ada, in mind. Compared to older design methodologies that emphasize either data or functionality, OOD places equal emphasis on data and functionality (objects and operations). OOD also groups objects and operations in well defined, loosely-coupled, and cohesive partitions that define interfaces. This grouping, loose-coupling, and partitioning correspond to the software engineering concepts of abstraction, information hiding, and modularization. Ada features such as packages and tasks; types, private types, and limited private types; and programming unit specifications and body separation are consistent with OOD and software engineering concepts.

The main steps in traditional OOD are: (1) Defining the Problem, (2) Developing an Informal Strategy, and (3) Formalizing the Strategy.

The "Defining the Problem" step is divided into stating the problem in a simple English sentence, and analyzing and clarifying given problem information.

The "Informal Strategy" is then described in an English paragraph. The paragraph should *contain 7±2 simple sentences, and should be* written at a fixed level of abstraction.

"Formalizing the Strategy" requires grammatical analysis of the "Informal Strategy". This process involves identifying solution objects and types (nouns), solution operations (verbs), object and operation attributes (adjectives and adverbs), and groupings of objects and operations.

Interfaces of high-level, stand-alone program units are then determined; and computer code (specifications) for the interfaces is written. Stepwise refinement of the highest-level program unit is subsequently done until *resulting subprograms can be directly translated* into code; this process is then applied to the other stand-alone units at the current level of abstraction.

In traditional OOD, the methodology should be applied recursively to any operation implementation of 200 or more lines of code. This may lead to top-down problem decomposition in terms of decreasing levels of abstraction of objects and operations.

FUNCTIONAL DECOMPOSITION USING SADT

Traditional OOD is not designed for performing a front-end system analysis and decomposition. A SADT-based approach for performing a functional decomposition of the system may be utilized. This process of front-ending the OOD methodology is recommended by Booch and EVB.

The objective is to decompose the system into partitions that functionally represent the aircraft systems. All inputs to each partition are then determined, which when collated would provide the outputs from each partition. To support this process. A SADT-based tool, such as *Automated Structured Analysis Program* (ASAP)TM (5) may be utilized.

PROBLEMS ENCOUNTERED WITH SADT

Several problems resulted when applying the SADT process, mainly due to the restrictions imposed by the process in limiting the inputs and outputs between partitions. The functional elements are tightly-coupled which add to complications.

A fundamental problem of performing a functional analysis is that a fairly detailed evaluation of each partition is required to identify the interfaces between partitions. The information generated by this analysis can be large, particularly with tightly-coupled systems. This forces the designer to begin handling reasonably large amounts of data rather than studying the structure of the system solution. Further, this process conflicts with software engineering principles and the philosophy of the OOD approach.

TM ASAP is a trademark of Thomson-CSF, Inc.

A solution to this problem is to expand the OOD process and begin the system analysis process by giving consideration to system requirements. This is more consistent with a system engineering approach to the problem. The significance of this approach is that appropriate consideration must be given to the manner in which system requirements are derived and specified.

PROBLEMS IN USAGE OF TRADITIONAL OOD

In applying traditional OOD, three main problem areas have been determined; (1) the identification of objects, operations and groupings after the writing of the informal strategy, rather than before; (2) the confusion caused due to the ambiguity of English syntax and semantics; and (3) the implied requirement that coding be done early in the design process.

The steps in traditional OOD require that the Informal Strategy be written prior to the listing of objects, operations, and groupings of objects and operations. This order seems to be the reversal of the ordered thought process used when OOD is actually applied. The objects, operations, and groupings are typically determined from the analysis and clarification of information given at the start of the design process, and must be in the software engineer's thinking before writing the informal strategy. This strongly implies that the objects operations, and groupings should be determined before the informal strategy is written.

The use of English syntax and grammar in traditional OOD is likely to cause confusion, due to ambiguities regarding noun, verb, and modifier identifications and meanings. This confusion contributes to communication, management, and quality assurance problems based on English, and not actually related to the substance of a project. If those involved in an application area have developed area-specific precise forms of description and meaning, these forms should be considered as replacements for English within OOD. Diagrams, graphs, charts, outlines, summaries, etc. are often more directly meaningful and precise than English text.

Traditional OOD forces consideration of code too early in the software development process. In Defining the Interfaces, and possibly before, programming language units and their specifications (code) must be considered. And, in implementing the solution, coding of specifications and program unit bodies must be done before possible further design involving recursion of OOD. This will likely clutter and hinder the design with implementation details rather than allow the implementation details to be put off until a later coding phase of the software development. The usual sequence of software development phases has design before coding, and does not mix the two directly; this allows the solving of a problem based on a few, easy-to-grasp, high-level partitions in the design; before implementing the solution based on many, hard-to-consider, low-level details in code.

SOLUTIONS TO PROBLEMS OF TRADITIONAL OOD

The traditional OOD usage problem; related to ordering the writing of the Informal Strategy before the listing of objects, operations and groupings; can be solved by reversing this ordering. The reversal is consistent with what a software engineer actually does in applying OOD. It is only after an engineer has abstracted objects, operations and groupings from already available information, that he can write them into an Informal Strategy.

The proposed solution requires greater emphasis on analysis and clarification of given information than in traditional OOD, and requires the determination of objects, operations and groupings before specifying the Informal Strategy. The Informal Strategy becomes a summation of an analysis and clarification of givens that includes listings of objects, operations and groupings.

Implementing this solution greatly aids the solving of traditional OOD usage problems that are associated with English language ambiguities. Since objects, operations and groupings are determined before the Informal Strategy is written, there is no need to determine them by highlighting the nouns, verbs and modifiers. The use of diagrams, graphs, charts, or outlines is emphasized and the use of English for software analysis and design is de-emphasized.

OOD can lead to premature coding, due to the point at which recursion is done in the traditional OOD sequence of steps. This point is at the end of the entire sequence and forces coding in one sequence before design when recursion occurs. This problem can be solved by positioning the call for recursion after the Informal Strategy step and before the Formalization of Strategy. The design can then be done through successive recursions of the first two OOD steps, before coding is begun.

These proposed solutions more fully support the use of software engineering concepts of abstraction, information hiding and modularization in OOD. The determination of objects, operations and groupings, before the writing of the Informal Strategy, allows the abstraction of the objects, operations and groupings in partitions (modules) when they are needed, rather than after the Informal Strategy. The repositioning of the call for recursion aids abstraction, information hiding, and modularization, by aiding progressive abstraction and expansion from high-level (design level) to low-level (code-level); and by encouraging that low-level details (code) be hidden within partitions (modules) during design.

The three proposed solutions require alterations to the traditional OOD steps. These alterations, together with additional proposed steps, are listed in Table 1. The steps which are new or re-ordered, compared to traditional OOD, are discussed in the following paragraphs.

Table 1. Burtek Refined OOD

A. <u>DEFINE THE PROBLEM</u>

1. STATE THE PROBLEM (IN AN ENGLISH SENTENCE)
2. ANALYSIS AND CLARIFICATION OF GIVEN INFORMATION (INCLUDING PROJECT REQUIREMENTS)
    a. Determine the Level of Abstraction of the Analysis
    b. Specify Assumptions and Limitations
    c. Identify Objects and Object Attributes
    d. Identify Operations and Operation Attributes
    e. Group Objects and Operations in Partitions (Consider Inclusion and Dependency)
    f. Identify Partition Interfaces (Use Tabulation and Graphics to Document Groupings, Dependencies and Interfaces)
    g. Identify Partitions that require further Decomposition

B. <u>SPECIFY THE INFORMAL STRATEGY (A DESCRIPTION OF THE SOLUTION BASED ON ANALYSIS AT THE CURRENT LEVEL OF ABSTRACTION)</u>

(VALIDATE THE INFORMAL STRATEGY)

****FOR EACH PARTITION REQUIRING FURTHER DECOMPOSITION
RECURSE THE METHODOLOGY FROM START TO HERE****

C. <u>FORMALIZE THE STRATEGY</u>

1. DEFINE THE INTERFACES (IN TERMS OF ADA MODULES AND DEPENDENCIES CORRESPONDING TO NONDECOMPOSED PARTITIONS)
2. IMPLEMENT THE SOLUTION
    a. Use the Stepwise Refinement to Optimize Modularity
    b. Determine ADA Specifications for ADA Units
    c. Implement the ADA units

DETERMINE THE LEVEL OF ABSTRACTION

This new OOD step explicitly indicates the importance and placement of level of abstraction determination and also allows OOD to better support the software engineering concept of abstraction and related information hiding.

Each analysis and clarification of givens is done in a context that includes a comprehensive network of groupings of objects and operations, covering the problem solution space relative to an acceptable granularity of space, time, logical complexity as well as other selected criterion. This context is called level of abstraction.

A level of abstraction is determined for each recursion of OOD. The problem statement for a recursion bounds a space (establish a scope) to be covered by the level of abstraction, and the determination of this level precedes and influences the steps that follow in the recursion. Levels of abstraction must be carefully determined by software engineers and supervisors.

SPECIFY ASSUMPTIONS AND LIMITATIONS

This new OOD step specifies the main assumptions and limitations of the current level of abstraction. This information allows the customer and the software engineer to have a common understanding before design continues.

IDENTIFY OBJECTS, OPERATIONS AND GROUPINGS

These three steps are not new, but are newly placed and accomplish different results compared to traditional OOD, as discussed in SOLUTIONS TO PROBLEMS OF TRADITIONAL OOD.

IDENTIFY PARTITION INTERFACES

This is a new OOD step, not equivalent to the traditional Defining the Interfaces step, that comes later. This step emphasizes the importance of design-level partitions and their dependencies and interfaces, rather than the Ada program units and interfaces to be considered in later steps.

The documentation for this step, and the immediate predecessor, includes a graphic presentation showing inclusion, interfacing, and dependency. This documentation shows how the previous steps, in the current recursion, support the software engineering concepts of abstraction, information hiding, and modularization by graphically revealing the partitions (modules) abstracted, and by hiding other information.

IDENTIFY PARTITIONS FOR FURTHER DECOMPOSITION

Software engineers and supervisors should identify partitions at the current level of abstraction that decompose into new object-operations partitions at the next level of abstraction (in the next recursion of OOD). Although length or complexity of projected implementation of a partition may be a meaningful determinant for further partitioning, the experience of engineers and supervisors in the project application area will be the major determinant.

SPECIFY THE INFORMAL STRATEGY

This step is equivalent to Developing an Informal Strategy, in traditional OOD. The sequential placement is after identifying objects, operations and groupings, rather than before.

VALIDATE THE INFORMAL STRATEGY

This new step encourages evaluation of the effectiveness of the Informal Strategy as a summary of the results of the steps in the current recursion. A valid Informal Strategy is written according to the current level of abstraction, and related assumptions and limitations; and should reflect objects, operations, attributes, groupings and interfaces.

The validation process may require that changes be made in the results of preceding steps in the current recursion. Such changes are elaborated in other design steps that the changes influence.

RECURSIVE APPLICATION OF THE METHODOLOGY

The importance of recursing OOD, based on placement of this step, is discussed in SOLUTIONS TO PROBLEMS OF TRADITIONAL OOD.

FORMALIZE THE STRATEGY

The substeps in this main step are essentially the same as those in Defining the Interfaces and Implementing the Solution, in traditional OOD. The main exception to this has to do with the fact that recursion is no longer done as in the traditional OOD sequence, as previously discussed in SOLUTIONS TO PROBLEMS OF TRADITIONAL OOD.

SUMMARY

This paper has presented an approach for developing simulator software using Ada. The approach has been derived from an existing methodology that has been expanded and refined to provide a system engineering solution. The refined methodology is object oriented and transforms system requirements into a software system. The changes to the existing methodology include refinements to better identify system partitions and components that become Ada units. In addition, the methodology has been integrated with the system engineering process of establishing requirements and solutions, that leads to Ada software. Booch(6) has significantly altered the OOD process. The modifications are essentially similar to Burtek's refinements to the OOD process and supports an object oriented development process.

REFERENCES

(1) Booch, Grady; <u>Software Engineering with Ada</u>, Menlo Park, CA; Benjamin/Cummings, 1983

(2) EVB Software Engineering Inc.; <u>An Object Oriented Design Handbook for Ada Software</u>

(3) Ron, Douglas & Brackett, John W.; <u>An Approach to Structured Analysis</u>, Softech, Inc.

(4) Michael Jackson; <u>System Development</u>, Prentice-Hall International Series in Computer Science

(5) Thomson-CSF, Inc.; ASAP Documentation package, 5350 Hollister Ave, Suite C, Santa Barbara, CA 93111

(6) Booch, Grady; "Object Oriented Design", IEEE Transactions on Software Engineering, Vol SE-12, No. 2, February 1986, p. 211-221

ABOUT THE AUTHORS

<u>Dr. Matt Narotam</u> is a staff engineer at Burtek, responsible for the technical performance of the Ada Simulator Validation Program. Dr. Narotam holds a Ph.D from the University of Salford, England, for a Thesis on Continuous Systems Simulation Language (CSSLs) implementation on mini-computers. Prior to joining Burtek, Dr. Narotam held the position of Research Scientist at the Computer Simulation Center. University of Salford, where he investigated simulation techniques for CSSLs. At Burtek, Dr. Narotam led a team of systems engineers in the development of training equipment for the F/A-18 aircraft. Prior to his appointment as staff engineer, Dr. Narotam was Supervisor, Software Systems.

<u>Dr. Sabina Saib</u> is the Technical Director of the Santa Barbara operation of THOMSON-CSF, Inc. Dr. Saib is a consultant to Burtek for the ASVP. Dr. Saib has been researching Software Engineering Methodology and tools for performing structured analysis and design techniques. Dr. Saib has also been evaluating automatic test tools. This work is being performed with respect to Ada application to software engineering. Dr. Saib has also been closely involved with real-time and multi-tasking issues with Ada. Dr. Saib has authored several book and papers on Ada and has served as joint editor with Robert E. Fritz in a book titled "The Ada Programming Language: a Tutorial". This document contains several papers covering real-time and multi-tasking as applicable to embedded computer system applications.

<u>Mr. Clifford Layton</u> is Director of the Computer Science Division at Rogers State College (RSC), Claremore, Oklahoma. Mr. Layton teaches several courses at RSC on Ada and software engineering. Mr. Layton has a close working relationship with Burtek on the ASVP and has been involved in Burtek's research and development program investigating Ada real-time and multi-tasking issues. Mr. Layton is a member of several professional bodies and is chairman of the Oklahoma Special Interest Group on Ada (SigAda) which is affiliated with the Association for Computing Machinery. Mr. Layton holds a Degree in Computer Science and Mathematics.