THE BENEFITS OF BOTTOM-UP DESIGN

Gregory McFarland

Grumman Data Systems
1000 Woodbury Rd.
Woodbury, N.Y. 11797

ABSTRACT

This paper examines an inconsistency in generic 'top-down' design methods and standards employed in the implementation of reliable software. Many design approaches adopt top-down ordering when defining the structure, interfaces, and processing of a system. However, strict adherence to a top-down sequencing does not permit accurate description of a system's error handling functions. The design of a system's response to errors is becoming critical as the reliability requirements of systems increase. This paper describes how top-down methods such as Object Oriented Design and Structured Design do not adequately address the issues of error handling, and suggests using a bottom-up substep within these methods to eliminate the problem.

1. INTRODUCTION

This paper describes the inability of top-down design techniques to allow for accurate design of the error handling features of a system. The primary concern involves what is tentatively termed the 'detailed design phase' of the software development process. This is the portion of the design process which provides a description of the system used as input to the implementation phase of the software life cycle. We believe that this design must accurately describe all the intended operations of the system to avoid the risk of 'interpretation' by programmers. Our discussion will make it clear that strict top down design techniques do not provide the designer an opportunity to specify the error handling features of a system. Acknowledging the mounting interest in 'structured design methods,' we must be certain that these methods address all of our requirements as designers, and that adopting them would not preclude certain design decisions. Additionally, the heightened reliability requirements of our systems necessitate that design methods provide the opportunity to address error handling issues.

Section 2 describes the software design environment we are considering. Section 3 defines techniques, standards, and tools often applied in the detailed design phase. The software design process is investigated by examining the activities performed during that effort in section 4. Section 5 details the problem encountered when using generic top-down methods in relation to the design of error handling facilities. Finally, section 6 describes how a bottom-up substep can be incorporated into existing methods to eliminate the problem.

2. SOFTWARE DESIGN ENVIRONMENT

For purposes of this paper, we will adhere to the definitions for 'life cycle' and 'method' found in [MCDE84]. The software life cycle defines a series of system views, each progressing from the abstract to the more concrete. A development method is concerned with the activities on one or more of these levels and comprises three distinct pieces: notation, guidelines, and analysis. The guidelines define rules for transforming the system at the previous level to the system at the current level. The current level is expressed in the notation defined by the method. Analysis is used to verify consistency within a level as well as that between levels.

A software development effort includes selection of a method to be applied in each life cycle phase. As indicated in the introduction, we are primarily concerned with the 'detailed design phase' where a representation of the system that can be used as a baseline for the coding or implementation phase is produced. According to the above definition of 'method,' few design techniques described in the literature today are 'methods.' Quite often only guidelines and/or notation are defined. Analysis techniques are rarely included. Additionally, individual efforts will normally modify the notation used based on past experience and tool availability. For this paper, we will concentrate our attention on the guidelines portion of the method. Therefore, we will assume that the final notation of the system after this phase is some form of Ada* PDL, that a PDL processor or Ada compiler is utilized to verify internal consistency, and some sort of structured design review is employed to verify the correctness of the resulting design in relation to previous design phases. We do not preclude the use of graphics during the design process, or as an additional output, but it will be the PDL that the programming staff utilizes during the implementation phase, and therefore this will be the final design notation. The final PDL representation of the system typically will define the system's modular structure, its data, and the processing to be performed by each module.

* Ada is a registered trademark of the U.S. Government, AJPO

3. SOFTWARE DESIGN TECHNIQUES, STANDARDS, AND TOOLS

The software development process is a complex combination of techniques, standards, and tools. Techniques are defined by the selected method and dictate the design steps. Standards are often dictated by contracts and impose additional constraints on the process. Tools can be automated aids such as editors, or logical tools such as the use of abstraction or information hiding. The combination of the various techniques, standards, and tools involved in each part of the design process can lead to problems like those described below.

Many design techniques found in the literature impose a top-down order of work within the level or phase where applied. The examples we will discuss are Object Oriented Design [OBJE85] [BOOC83] and Composite (Structured) Design [MYER78]. Both of these methods are 'top-down' since they require recursive application of the technique on the modules or operations that were defined in the previous step. In the case of Object Oriented Design, once the objects and operations have been defined, the designer must define the interfaces to these operations, perform a stepwise decomposition of the highest level module, and then repeat the entire design process for the newly defined operations. The stepwise decomposition of the highest module defines the interaction of this module with the newly defined operations. The implementation of these operations is not considered; they are 'abstractions.' Structured design incorporates a similar set of tasks for the design process, the main difference being the rules (guidelines) used to define the modules that 'implement' the current module. In structured design, only the structure of the system is defined. No method for defining the algorithmic portion of each module is proposed. If the technique employed to define each module's implementation section applies a top-down approach, then the entire detailed design phase is considered top-down.

Additionally, DoD standards and guidelines [DOD] for developing software systems impose a top-down structure on the development process. Unless alternate development techniques are approved by the contracting agency (see [SDST85]), top-down design, top-down coding, and top-down testing are required. As will be argued in the remainder of this paper, the use of a top-down ordering of the entire detailed design process is not desirable.

Many design techniques, including the two above, employ 'abstraction.' Abstraction is a valuable tool of the software engineer, but will be shown to be inappropriate if used throughout the entire detailed design phase. Abstraction allows designers to ignore the implementation details of 'other' parts of the system. This is useful during a decomposition process, but will lead to problems when connected with the design of a system's error handling facilities.

We will see how the combination of the above three items, top-down design techniques, contractual standards, and the utilization of abstraction, leads to problems when designing the error handling facilities of a system. A bottom-up approach may be applied during one substep of the overall detailed design process to eliminate this problem.

4. SOFTWARE DESIGN PROCESS

Consider the activities that occur during a typical detailed design effort. The selected method defines a set of guidelines which describe the steps a designer must undertake during the design process. As stated above, the design at this level typically includes module definitions, their relationships with each other, data definitions, and a description of the processing each module should undertake. The generic top-down design techniques being considered proceed as follows. First, select an undefined module and follow the guidelines specified by the technique. These guidelines result in additional modules and data definitions being defined. Second, determine the interfaces of these new modules and data objects. The guidelines may then suggest one of two possibilities. In the case of Object Oriented Design, stepwise refinement or some other technique is adopted to define the processing of the module. Once this is accomplished, the method is recursively applied to any resulting modules too large to be described as a single unit. An alternative approach, which might be found in a Structured Design, would be to first repeatedly apply the method to any undefined modules, completely defining the modular structure of the system and the interfaces to these modules. Once the entire system is decomposed, each module's processing is described, most likely in a top-down order.

Abstraction plays a large role in these top-down techniques. Abstraction permits the designer to utilize the interface information of other modules in the design of any module's implementation section. A hierarchy of modules is often viewed in a top-down fashion, with each module taking an abstract view of lower level modules in its 'implementation section.' The application of abstraction implies that only the interface information is needed for correct use of a module. Top-down implies that interface information for any module is used prior to that module having its implementation section defined. Thus we are relying on the premise that the design of any implementation section will not alter the interface of a module. In the case of error handling, this may not always be true.

5. THE PROBLEM

The problem associated with top-down design techniques and the use of abstraction becomes evident when considering the design of a module's processing section. This design will utilize prior design work that has identified interfaces and functionalities of subordinate modules. In other words, this processing section's design is based on the abstractions provided by the subordinate modules. Thus, the correctness of this design relies on the premise that these interfaces or abstractions will not change. While change is a natural part of the design process, attributable to designers' discovery of new information and backtracking to modify prior design decisions, change and backtracking should not be a direct consequence of the method used. Two assumptions concerning the error handling facilities of a module, which will be justified below, are that these facilities will not be known until the module's implementation is designed, and that these facilities will change the interface of the module. Based on these two assumptions, the design of every implementation section may change the associated interface. Therefore, the design of the processing section described above may become invalid when the subordinate modules' processing sections are defined. Since a top-down order of design is being employed, every processing section that causes changes in the associated interface, invalidates the assumptions used to design the processing section of superior modules.

First, the assumption that the error handling facilities of a module will change that module's interface should be considered. Errors can not be handled entirely within the module where they are generated. If errors were always handled locally, either no real error processing or correction would be performed, or each module would require knowledge of its actual use or purpose. Thus, either the systems will not be tolerant of errors, or the individual software within the system will not be general or reusable. For these reasons we will allow and even encourage that errors be propagated from modules and be handled where it is most appropriate. Now consider that a complete design, at the detailed level, will specify the potential error situations as well as the desired response to those errors. Errors may be propagated into or generated by a module. Depending on the error handling facilities provided in the chosen language, errors may or may not be gracefully handled. Consider the Ada programming language which provides extensive error handling facilities. In Ada, errors may be handled by special sections of code, and propagated out of the current module. The processing performed in response to errors changes the functionality or effect of this module. The possibility of errors being propagated out of a module also changes the interface of the module. Thus, the error handling facilities of a module add to or change the module's interface.

Consider also when the designer will be making decisions about the error handling of some module. Abstraction plays an important role in the application of the design method. Modules are defined in terms of their function and interface, while their implementation is not considered. These modules are then utilized during the design of the processing sections of superior modules. During the definition of a module's function and interface it is possible to define certain error situations that may arise. However, defining the internal response to these errors would imply that the designer is considering the implementation details of the modules. This is a violation of the abstraction principle and is inappropriate. Additionally, designers can not be cognizant of all the possible errors a module may generate. These errors will be discovered during the design of that module's implementation section. Accordingly, at the outset, the response to these errors will also be unknown. Therefore, there is a considerable potential that the interface of a module will be changed after that interface has been defined and used during earlier design activities.

The basic flaw described above is a consequence of the designer's reliance on the abstractions of other modules. The principle of abstraction has proven very useful in defining the structure of a system. However, it generally does not apply to the entire design process. It is unwise to design the implementation section of a module based on a number of abstractions if there is a likelihood that the abstractions will change. Doing so creates the potential for considerable rework and deviation from contractual standards and procedures.

The assumption made above that "a complete design, at the detailed level, will specify the potential error situations as well as the desired response to those errors," should be discussed. The content of a detailed design is a subjective decision. The life cycle phase considered in this paper, labeled 'detailed design,' was more accurately defined as the phase prior to implementation. Thus, the output of this phase, a description of the system in the selected notation, will be given to a programming staff for purposes of implementation. Alternatives to the above assumption are to not specify the error handling facilities to be incorporated by the system, or to specify them only partly. Consequently, the programmer must decide between not including any error handling facilities since they were not defined, or in the case of Ada, providing a general error handler that catches any error raised in or propagated to a module. Neither of these situations is desirable if reliability is a goal of the software. Alternatively, the programmer may handle those errors which he determines are generated by this module on an individual basis, deciding what processing is appropriate for each, and which should be propagated to calling modules. This will cause a module's implementation to deviate from its assigned function and interface. Finally, the programmer may perform the necessary work to make the following determinations:

1. Which errors may be propagated into the module?

2. What processing has already been performed in response to these errors?

3. What errors may be generated by this module?

4. What processing is necessary in response to both types of errors? and

5. Which errors get propagated out of this module.

This alternative requires communication between programmers and additions to the functionality and interface of the modules. None of these alternatives is as attractive as having the error handling facilities defined during the design process.

6. A SOLUTION

A simple solution to this problem is to design the processing sections of a system's modules in a bottom-up order. As each module has its processing section designed, appropriate changes can be made to the interface and functional description of the module. Thus, higher level modules utilize a more complete description of lower level modules. Performing this bottom-up substep within a design phase is compatible with both Object Oriented Design and Structured Design. This substep only requires that implementation sections are not designed until the structure and data definitions of the entire system have been defined. Once this is accomplished, the bottom-up order of processing section design may begin.

A bottom-up design order does not define any additional guidelines for the design of the error handling facilities of a system. At most, this will allow the designer the opportunity to consider the issue, and specify the required functionality prior to when that information is used in other design work. This will reduce the amount of change and wasted effort that results from basing design decisions on incomplete information.

7. SUMMARY

This paper defines a problem engendered by the top-down structure imposed by software design methods and standards applied during the detailed design of a software system. Designers whose techniques rely on abstract modules defined in a top-down order will find that the design of the implementation section of these modules will result in changes to their interfaces attributable to error situations defined,

handled, and propagated. Changes to these interfaces invalidate assumptions made by higher level modules' implementation sections. One solution is to design modules' implementation sections in a bottom-up order, making the necessary changes to the interfaces of the modules.

This paper is not meant to criticize current methods or imply that they should be abandoned. Instead, it criticizes the ways in which these methods are applied. What is desired is an understanding that application of 'design methods' does not solve all the problems of software design. In addition to being executed correctly, design methods must be applied only where appropriate. Careful analysis is needed to determine what must be accomplished during each phase of the software life cycle, and how well the selected method or methods address these needs. It will often be found that existing design methods can not address all the activities required within even a single phase of the life cycle. For this reason, methods must be augmented with additional techniques or considerations to ensure the design process is complete and correct. The example described in this paper demonstrated that the design of error handling facilities of a system is not adequately addressed by generic top-down methods. Thus, special consideration is required to ensure that the overall design approach addresses this portion of the software system.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the encouragement and helpful comments of John D. Litke.

REFERENCES

[BOOC83] Booch, Grady; Software Engineering With Ada; Benjamin/Cummings Publishing Company, Inc., California, 1983.

[DOD] DOD-HBK-287 Defense Department Software Development Handbook

[DOD] DOD-STD-2167 Defense System Software Development

[MCDE84] McDermid, John, and Ripken, Knut; Life Cycle Support in the Ada Environment; The Ada Companion Series; Cambridge University Press, Great Britain; 1984.

[MYER78] Myers, Glenford, J.; Composite/Structured Design; Van Nostrand Reinhold Company, New York; 1978.

[OBJE85] Object Oriented Design Handbook; EVB Software Engineering Inc., 1985.

[SDST85] SDS - Tailoring Provisions for Ada; Technical Report E-98044; Electronic System Division, Air Force Systems Command, Hanscom Air Force Base, Bedford, MA, April 1985.