A COMPARISON OF THE RADICAL AND CONSERVATIVE METHODS OF TOP-DOWN SOFTWARE DEVELOPMENT

Greg Hill

IBM Corporation, General Products Division, Tucson, Arizona 85744

Abstract

This paper describes the conservative and radical approaches to top-down software development and makes a comparison of the two methods. This comparison discusses the advantages and disadvantages of each method and shows how they differ in affecting the development costs, efficiency, reliability, and maintainability of the resulting software. This paper also describes a hybrid approach to software development, which has successfully been used to obtain many of the benefits from both the conservative and radical methods of top-down software development.

Introduction

The term top-down software development means different things to different people. This paper discusses two divergent methods of software development which, although very different, are both categorized as top-down.

The top-down software development philosophy emerged many years ago as a technique for avoiding problems by placing controls over the ordering of steps in the software-development process. Since then, much has been written about the success of top-down design, top-down coding, top-down testing, and so forth, for example, see [refs. 1 - 7]. Although there is seldom any mention in the literature as to whether the conservative approach or the radical approach has been used to derive the successes in applying the top-down software development philosophy, Yourdon [ref. 8] does provide an assessment of the circumstances as to when each method should be used.

Following this introduction is a description of the conservative approach to the top-down software development philosophy. Then comes a description of the radical approach, followed by a comparison of the two approaches, and a description of a hybrid approach which has been used in actual practice to achieve quite favorable results.

To simplify the later discussions in this paper, assume that a software program or system is to be developed which, when completed, will have the structure shown in Figure 1 below:

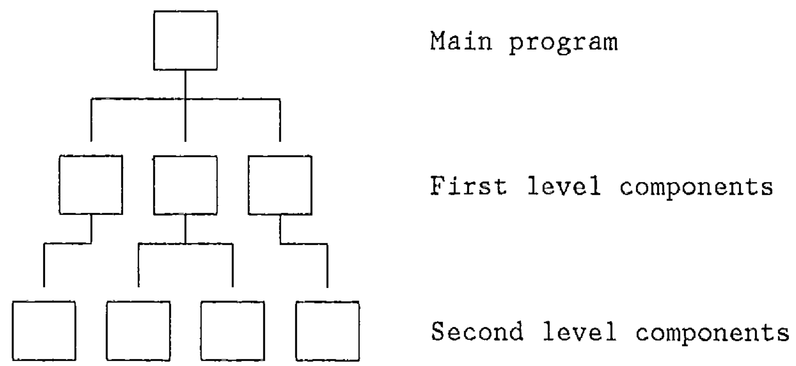

Figure 1. Sample Program Structure or System Structure

Description of the Conservative Method

The conservative method requires that the design be done in a top-down fashion, and that it be completed prior to the start of any coding. The coding phase is then done in a top-down fashion, and it is completed prior to the start of any testing. Finally, the testing phase is completed in a top-down fashion.

This conservative approach to top-down software development can be shown in Figure 2 below,

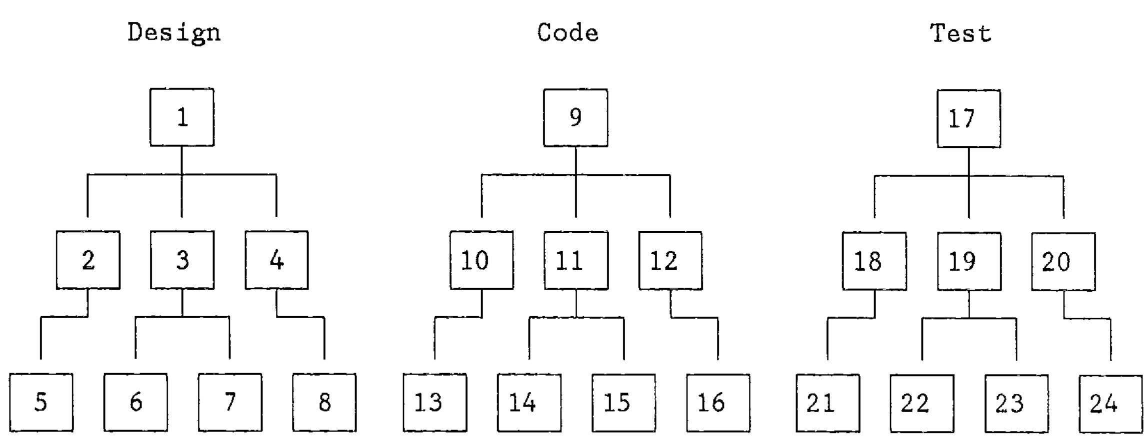

Figure 2. The Conservative Approach

where the numbers represent the sequence in which development activities are performed on the individual components, regardless of whether the components are subroutines of a larger program or modules in a larger system.

Description of the Radical Method

The radical method requires that designing, coding, and testing all be done in a top-down fashion, but each component be completely designed, coded, and tested prior to starting the design of the next component.

The radical approach to top-down development can be shown in Figure 3 which is similar to that for the conservative method, but with a different numeric ordering of activities.

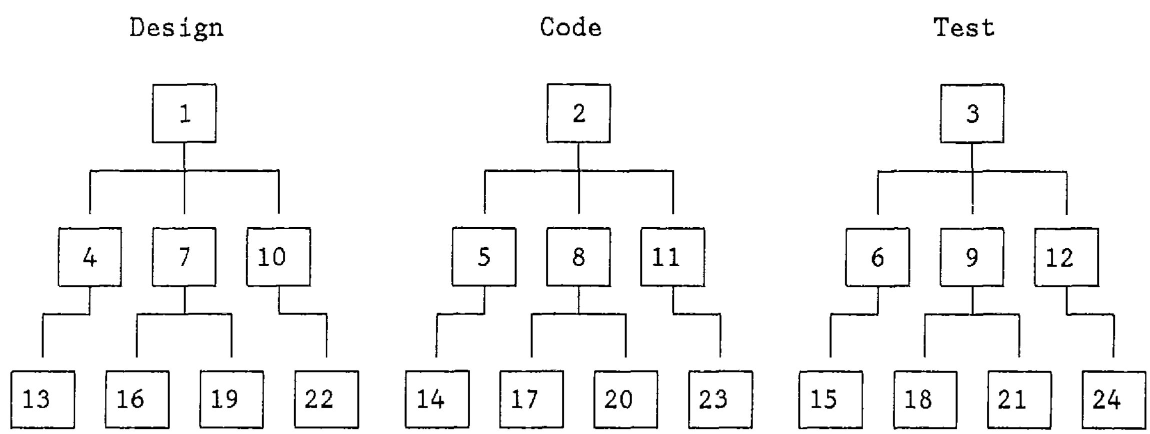

Figure 3. The Radical Approach

Note that the numbers represent a radically different sequence for performing the development activities on the program or system components.

Both the conservative and radical approaches represent the top-down philosophy of software development, yet they have a different impact on the resulting software. We will now take a closer look at the effects of each approach on the software-development activities and on the resulting software itself.

Effects on Design

The conservative approach results in the overall design being completed at an earlier time than with the radical approach. This is advantageous in that it allows designers to document a complete design in preparation for formal design reviews, prior to coding. It is also advantageous in that it provides the information needed to construct coding and testing schedules, resource estimates, and budgets.

The radical approach has the advantage of detecting minute design flaws that may be easily overlooked in a formal design review process. The reason for this is that the subsequent coding and testing activities, that is, compiling, linking, and exercising the code which is representative of a portion of the program or system designed thus far, are likely to uncover any remaining design flaws. The radical approach has this advantage because a computer is faster than a person at compiling, linking, and exercising code, and because these activities take place at an earlier time with the radical approach than with the conservative approach.

A disadvantage of the radical approach is that because the overall design is not completed until a much later time than with the conservative approach, it can be difficult to put together an accurate set of schedules, budgets, and resource requirements early enough to be useful.

Effects on Coding

In the conservative approach, the complete design is available to programmers before coding starts. This allows programmers to gain a better understanding of the big picture of what the program or system will do before they start coding. It also allows the users to gain a better understanding of what they have asked for. In case they wish to make changes at this point, there is no code to throw away; however, the design will have to be adjusted accordingly, and should also be reviewed again.

With the radical approach, high-level components will be implemented and tested at an earlier time than with the conservative approach. This allows programmers to have something to show for their design efforts, and gives managers something more tangible than design to measure. In case users wish to make changes at this point, there may be some code thrown away if the code represents a high-level component that has already been designed and coded. However, if the changes are to components that have not yet been designed, the changes can be designed into the program or system on the first pass.

Effects on Testing

Because the conservative approach results in the entire program or system being completely coded prior to any testing, it greatly extends the necessary testing effort by requiring that the top-down testing operate on a complete program or system, or that the lower-level components be temporarily exchanged for component stubs. The first choice increases the amount of code that will need to be analyzed as the potential cause of any bugs found in testing, while the second choice will increase the amount of code-editing activity needed to swap components in and out with their associated stubs.

On the other hand, the radical approach to top-down software development results in the testing of each component prior to the existence of the lower-level components. This means that the lower-level components will have to have stubs inserted into the code to do top-down testing. These stubs will then be replaced with the real components when the coding stage is reached for the lower levels in the program or system.

As far as top-down testing is concerned, the conservative approach results in either an all-at-once testing effort or a high-code-movement testing effort, while the radical approach results in a piecemeal testing effort that more easily identifies the component containing the bug.

Yourdon [ref. 8] notes many advantages of top-down testing in a comparison with bottom-up testing. Based on the conservative versus radical discussion above, it appears that Yourdon's top-down testing discussion relates to the radical rather than the conservative approach. The advantages that he mentions for top-down testing, which happen to be the advantages of the radical top-down approach, are listed below:

- Major interfaces are exercised at the beginning of the project
- Users can see a working demonstration of the system at an early stage
- Deadline problems can be dealt with more easily
- Debugging is easier
- Requirements for machine test time are distributed more evenly throughout the project development
- Programmer morale is improved
- The need for test harnesses or scaffolding is eliminated

Effects on Efficiency, Reliability, and Maintainability

Because the radical approach provides a working subset of the program or system at an earlier time than the conservative approach, any significant performance problems can potentially be recognized and corrected at an earlier stage in the development. In the conservative approach, it is much more difficult to recognize where efficiency problems will occur in a program or system by looking only at its design.

Because top-down testing is essentially system-integration testing for that portion of the program or system that is already designed and coded, the higher-level components will be repeatedly exercised in both the conservative and radical approaches, thus improving the overall reliability of the software. However, the conservative approach does not allow the system integration testing to start as early as in the radical approach. Because of this, the radical approach can lead to improved reliability over that resulting from the conservative approach.

Maintainability of the resulting program or system can be greatly influenced by the number and size of changes requested by users during the coding and testing stages of development. In the conservative approach, these changes can be worked into the existing design and the new design recoded, or they can be applied to the code like bandaids or patches. The first method results in a more maintainable program or system at the expense of significantly delaying the completion of the project. The second method results in a faster adaptation of the program or system to changing user needs, however it does this at the expense of significantly reducing the future maintainability of the code.

In the radical approach, the changes are likely to be requested at an earlier time, particularly if they are based on the users involvement or experience with an early working subset of the program or system. The earlier that the changes are requested, the more likely it will be that the changes can be incorporated into the existing design without the degree of effort needed by the conservative approach.

A Hybrid of the Two Approaches

It appears so far that the conservative approach has more advantages in the design stage of top-down software development and the radical approach has more advantages in the later stages of top-down software development.

To gain the benefits of both approaches, we can divide the design stage into a coarse design and a detailed design stage. We can then perform the conservative top-down approach on the coarse design, followed by the radical top-down approach on the detailed design, code, and test stages.

Note that we have not eliminated steps, that is, steps 9 through 16 in Figure 4 each contain three steps in the earlier figures. We have simply partitioned the development work a little differently by doubling the number of steps needed to complete the design, and at the same time reducing the amount of effort needed to complete each of the design steps.

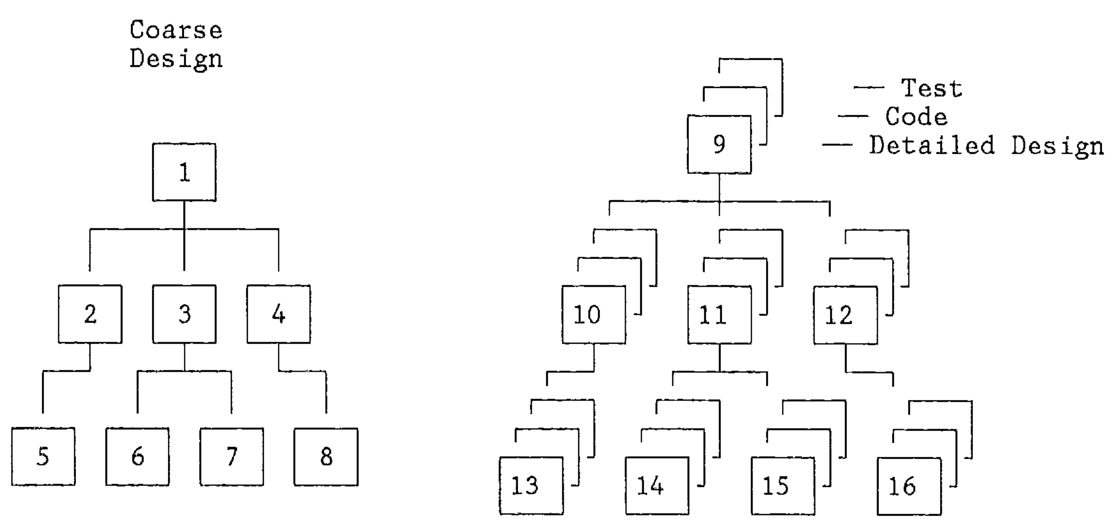

Figure 4. A Hybrid Approach

Such a hybrid approach has successfully been applied to the development of a variety of shared database applications and software tools by the Microcode Tools Engineering Center at IBM in Tucson.

The advantages of this hybrid approach are:

- At the completion of the coarse design, that is, step 8 in Figure 4, sufficient information has been gathered to:

    1. Prepare a design document for review
    2. Prepare a schedule for the detailed design, code, and test stages of development
    3. Determine resource requirements

- At the completion of the detailed design, code, and test of the highest-level component, that is, step 9 in Figure 4, a sufficiently tangible subset of the program or system exists to:

    1. Show users a part of what they asked for
    2. Show management that some of the design is successfully operating
    3. Serve as a building block for subsequent development, that is, steps 10 through 16 in Figure 4.

Conclusion

The top-down philosophy of software development can be divided into a conservative and a radical approach. There are advantages and disadvantages in using either approach for developing software.

The hybrid approach discussed in this paper permits all of the advantages of the conservative approach that are relevant to having the design work completed up front. It also permits all of the advantages of the radical approach that deal with the progression from design to code to test, on a component by component basis, in a top-down fashion.

This hybrid approach eliminates the primary disadvantage of the conservative approach by allowing early coding and testing efforts to produce some early tangible results. It also eliminates the primary disadvantage of the radical approach by allowing enough design work to be completed up front so that the big picture of the development work can be better understood and managed.

References

1. G. D. Bergland, "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14, No. 10, Oct. 1981, pp. 13-37.

2. C. L. McGowan, and J. R. Kelly, Top-Down Structured Programming, Petrocelli, New York, 1975.

3. P. W. Metzger, Managing a Programming Project, 2nd Ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

4. H. D. Mills, "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55.

5. H. D. Mills, "Structured Programming: Retrospect and Prospect," IEEE Software, Vol. 3, No. 6, Nov. 1986, pp. 58-66.

6. P. Van Leer, "Top-Down Development Using A Program Design Language," IBM Systems Journal, Vol. 15, No. 2, 1976, pp. 155-170.

7. E. Yourdon, Techniques of Program Structure and Design, Chapter 2, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

8. E. Yourdon, "Top-Down Design and Testing," Managing the Structured Techniques, Yourdon Inc., 1976.