# Structured Design Chapter Summaries

## Chapter 1.

In this chapter, we have attempted to demonstrate that there are a number of important roles in the systems development process – primarily, business systems analysis (understanding the problem), computer systems design (designing the major architecture of a solution to the problem), and programming (putting the design into code). We have concentrated (and will continue to concentrate for the remainder of this book) on the role of design, because we feel it is a critical area that, in many organizations, has been ignored, or delegated to the wrong person (e.g., the junior programmer), or performed by someone incompetent to do the job (e.g., a systems analyst whose last *real* experience was with an IBM 650). We have emphasized that it is not sufficient to find just *one* design for a computer system; what we want is the *best* design given appropriate information about the technical objectives and constraints for the system

Even more, we want an organized methodology – a “cookbook” – that will help us develop “good” designs and discard “bad” designs as easily as possible. That, in a nutshell, is what “structured design” is all about: a collection of guidelines for distinguishing between good designs and bad designs, and a collection of techniques, strategies, and heuristics that generally leads to good designs – good in the sense of satisfying the common technical objectives and constraints found in commercial and scientific environments.

In the next chapter, we will examine some of the fundamental philosophies and concepts that form the basis of structured design. If it seems that we are placing extraordinary emphasis on philosophies and concepts, have patience: It is critically important if the subsequent chapters on techniques are to make sense. Indeed, we suggest that when you finish Chapter 21, you return to reread the first two chapters!

## Chapter 2.

In this chapter we have seen that we can generally minimize the cost of implementation, maintenance, and modification – three of the major technical objectives for current computer systems – by designing systems whose pieces are small, easily related to the application, and relatively independent of one another. We have also seen that structured design achieves this by focusing attention on proper *partitioning* of the application and by proper organization of the pieces of the system. We have also introduced some general design philosophies, such as the “rule of black boxes,” which are extremely basic and which will be dealt with later in the book.

Finally, we have seen that a number of our value judgements about the design of a computer system can be expressed by drawing analogies to human organizations. In addition to providing a convenient communications tool between designers, it allows us to draw upon the experience of several hundred years of studying human organizational structures – which, after all, are just another kind of system with many of the same properties as software systems.

## Chapter 3.

We have seen in this chapter that computer programs are systems, and that they can be analyzed in a variety of ways. The most important components of a program are its *statements*, and we can recognize an inherent structure in those statements; indeed, unless *we do* recognize the statement structure, our attempts to modularize a program by chopping it into pieces will be unsuccessful.

Most of the emphasis in this chapter have been on the definition of terms and concepts used throughout the rest of the book. In addition to defining terms that are not part of the average designer’s vocabulary – e.g., normal and pathological connections – we have attempted a careful, technical definition of words like module.

## Chapter 5.

Although it may seem that this chapter is heavy on philosophy and light on practical advice, the philosophy actually forms the basis for almost all of the practical advice in Chapters 6 and 7 – not to mention a large portion of the rest of the book. It is absolutely essential that the designer realize that the major cost of developing computer systems is the cost of debugging, which, in turn, is the cost of *human* error.

And, it is essential that we are aware of the limitations of the human mind when we design computer systems (or any other complex system). Unless we realize that the cost of systems development can be reduced by partitioning systems into smaller pieces, we will be limited to developing systems of 100,000 lines of code or less. On the other hand, there is nothing to be gained from partitioning a system into modules of one instruction each; at some point, the simplicity of each individual module is outweighed by the complexity of the intermodule interfaces.

## Chapter 6.

This chapter has introduced one of the most important criteria for judging the goodness of a design: *coupling*. The next chapter discusses a related concept known as cohesion; together, these two concepts form the central theory of structures design.

As we have seen, there are several factors that influence the coupling between modules: the type of connection, the complexity of the interface, the type of information that flows between the modules, and the binding time of intermodular connections. In addition, the use of “global” data greatly increases intermodule coupling. Attempts have been made to *quantify* the strength of various types of coupling, but it will probably be several years before such a quantitative measures are accepted within the data processing profession.

## Chapter 7.

From the discussions in this chapter, you should *not* conclude that all logical modules are bad, nor that editing and validation always should be distributed throughout a system; nor should you attempt to derive any other black-and-white rules of thumb. High cohesion is not “good,” nor is coincidental cohesion “evil.” Module cohesion is associated with *effective* modularity; it has certain predictable effects on transparency, programmability, ease of debugging, ease of maintenance, and ease of modification.

Other things being equal, these qualities will improve as cohesion is increased. This does not mean that losses will not be incurred in other areas. For example, the designer may be able to save CPU time or memory, simplify the data flow, divide the programming task more easily, or reduce apparent duplication of effort by using relatively uncohesive modular organizations. The designer may save design effort by using relatively uncohesive modular organizations. The designer may save design effort, too, since logical and temporal groupings are comparatively easy to identify and describe – while complete functional cohesion may require extensive analysis and study.

The obligation of the designer is to know the effects of varying cohesion – especially the cost in terms of modularity – and to be prepared to trade this off against potential benefits in other areas of interest. Unless he gains more in decreasing CPU time than he loses in achieving long-term viability of a program, for example, he must choose a more functional organization.