Tom DeMarco

Structured Analysis and System Specification

Yourdon, New York, 1978
pp. 1-7 and 37-44

STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION

by

Tom DeMarco

Foreword by

P.J. Plauger

YOURDON inc.
1133 Avenue of the Americas
New York, New York 10036

First Printing, March 1978

Second Printing, June 1978

Revised, December 1978

ISBN: 0-917072-07-3

CONTENTS

PART 5: SYSTEM MODELING

PART 6: STRUCTURED ANALYSIS FOR A FUTURE SYSTEM

PART 1

BASIC CONCEPTS

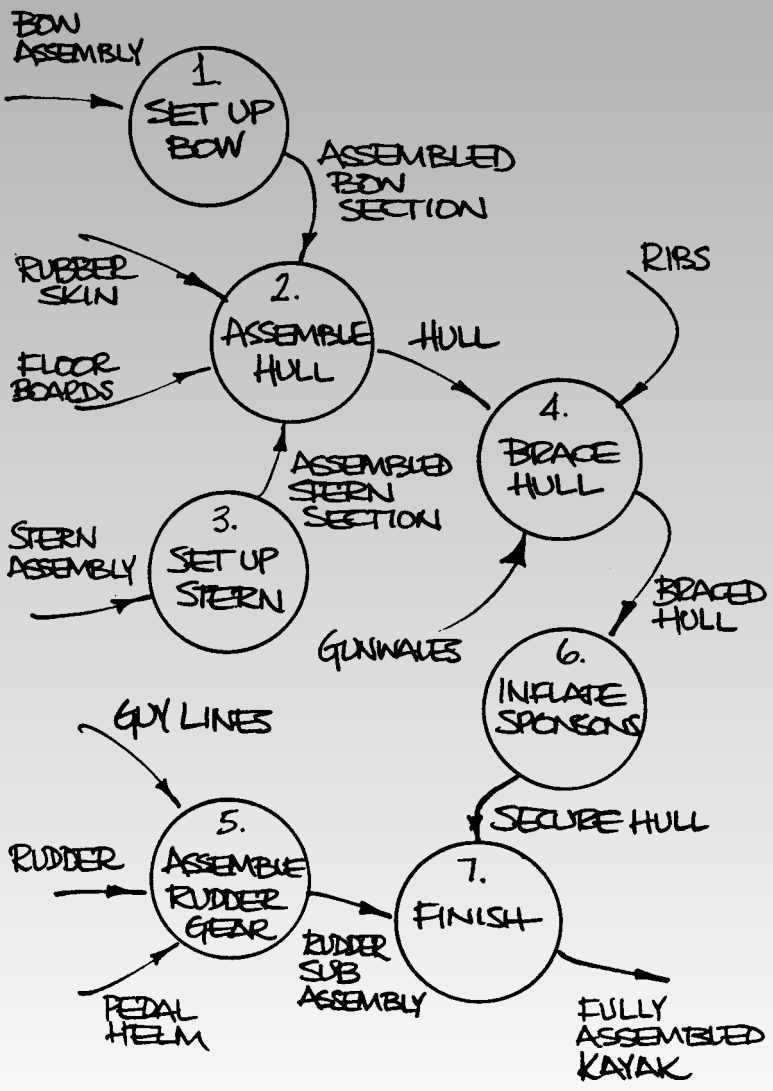
BOW ASSEMBLY
1. SET UP BOW
ASSEMBLED BOW SECTION
RUBBER SKIN
FLOOR BOARDS
2. ASSEMBLE HULL
HULL
RIBS
4. BRACE HULL
ASSEMBLED STERN SECTION
STERN ASSEMBLY
3. SET UP STERN
GUNWALES
BRACED HULL
6. INFLATE SPONSONS
GUY LINES
SECURE HULL
RUDDER
5. ASSEMBLE RUDDER GEAR
7. FINISH
RUDDER SUB ASSEMBLY
PEDAL HELM
FULLY ASSEMBLED KAYAK

Figure 1

1 THE MEANING OF STRUCTURED ANALYSIS

Let's get right to the point. This book is about Structured Analysis, and Structured Analysis is primarily concerned with a new kind of Functional Specification, the Structured Specification. Fig. 1 shows part of a Structured Specification.

The example I have chosen is a system of sorts, but obviously not a computer system. It is, in fact, a manual assembly procedure. Procedures like the one described in Fig. 1 are usually documented by a text narrative. Such descriptions have many of the characteristics of the classical Functional Specifications that we analysts have been producing for the last 20 years. (The Functional Specification describes *automated* procedures — that is the main difference between the two.) Take a look at a portion of the text that prompted me to draw Fig. 1.

Assembly Instructions for KLEPPER Folding Boats

1. Lay out hull in grass (or on carpet). Select a clean, level spot.
2. Take folded bow section (with red dot), lay it in grass, unfold 4 hinged gunwale boards. Kneel down, spread structure lightly with left hand near bow, place right hand on pullplate at *bottom* of hinged rib, and set up rib gently by pulling towards center of boat. Deckbar has a tongue-like fitting underneath which will connect with fitting on top of rib if you lift deckbar lightly, guide tongue to rib, press down on deckbar near bow to lock securely. Now lift whole bowsection using both arms wrap-around style (to keep gunwales from flopping down) and slide into front of hull. Center seam of blue deck should rest on top of deckbar.
3. Take folded stern section (blue dot, 4 "horseshoes" attached), unfold 4 gunwales, set up rib by pulling on pullplate at *bottom* of rib. Deckbar locks to top of rib *from the side* by slipping a snaplock over a tongue attached to top of rib . . .

   And so forth.

The differences are fairly evident: The text plunges immediately into the details of the early assembly steps, while the structured variant tries to present the big picture first, with the intention of working smoothly from abstract to detailed. The Structured Specification is graphic and the text is not. The old-fashioned approach is one-dimensional (written narrative is always one-dimensional), and the structured variant is multidimensional. There are other differences as well; we'll get to those later. My intention here is only to give you an initial glimpse at a Structured Specification.

Now let's go back and define some terms.

1.1 What is analysis?

Analysis is the study of a problem, prior to taking some action. In the specific domain of computer systems development, analysis refers to the study of some business area or application, usually leading to the specification of a new system. The action we're going to be taking later on is the implementation of that system.

The most important product of systems analysis — of the analysis phase of the life cycle — is the specification document. Different organizations have different terms for this document: Functional Specification, External Specification, Design Specification, Memo of Rationale, Requirements Document. In order to avoid the slightly different connotations that these names carry, I would like to introduce a new term here: the Target Document. The Target Document establishes the goals for the rest of the project. It says what the project will have to deliver in order to be considered a success. The Target Document is the principal product of analysis.

Successful completion of the analysis phase involves all of the following:

1. selecting an optimal target
2. producing detailed documentation of that target in such a manner that subsequent implementation can be evaluated to see whether or not the target has been attained
3. producing accurate predictions of the important parameters associated with the target, including costs, benefits, schedules, and performance characteristics
4. obtaining concurrence on each of the items above from each of the affected parties

In carrying out this work, the analyst undertakes an incredibly large and diverse set of tasks. At the very minimum, analysts are responsible for: user liaison, specification, cost-benefit study, feasibility analysis, and estimating. We'll cover each of these in turn, but first an observation about some characteristics that are common to all the analyst's activities.

1.1.1 Characteristics of Analysis

Most of us come to analysis by way of the implementation disciplines — design, programming, and debugging. The reason for this is largely historical. In the past, the business areas being automated were the simpler ones, and the users were rather unsophisticated; it was more realistic to train computer people to understand the application than to train users to understand EDP technology. As we come to automate more and more complex areas, and as our users (as a result of prevalent computer training at the high school and college level) come to be more literate in automation technologies, this trend is reversing.

But for the moment, I'm sure you'll agree with me that most computer systems analysts are first of all computer people. That being the case, consider this observation: Whatever analysis is, it certainly is not very similar to the work of designing, programming, and debugging computer systems. Those kinds of activities have the following characteristics:

- The work is reasonably straightforward. Software sciences are relatively new and therefore not as highly specialized as more developed fields like medicine and physics.
- The interpersonal relationships are not very complicated nor are there very many of them. I consider the business of building computer systems and getting them to run a rather friendly activity, known for easy relationships.
- The work is very definite. A piece of code, for instance, is either right or wrong. When it's wrong, it lets you know in no uncertain terms by kicking and screaming and holding its breath, acting in obviously abnormal ways.
- The work is satisfying. A positive glow emanates from the programmer who has just found and routed out a bug. A friend of mine who is a doctor told me, after observing programmers in the debugging phase of a project, that most of them seemed "high as kites" much of the time. I think he was talking about the obvious satisfaction programmers take in their work.

The implementation disciplines are straightforward, friendly, definite, and satisfying. Analysis is none of these things:

- It certainly isn't easy. Negotiating a complex Target Document with a whole community of heterogeneous and conflicting users and getting them all to agree is a gargantuan task. In the largest systems for the most convoluted organizations, the diplomatic skills that the analyst must bring to bear are comparable to the skills of a Kissinger negotiating for peace in the Middle East.
- The interpersonal relationships of analysis, particularly those involving users, are complicated, sometimes even hostile.
- There is nothing definite about analysis. It is not even obvious when the analysis phase is done. For want of better termination criteria, the analysis phase is often considered to be over when the time allocated for it is up!
- Largely because it is so indefinite, analysis is not very satisfying. In the most complicated systems, there are so many compromises to be made that no one is ever completely happy with the result. Frequently, the various parties involved in the

> negotiation of a Target Document are so rankled by their own concessions, they lose track of what a spectacular feat the analyst has achieved by getting them to agree at all.

So analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you.

1.1.2 The User Liaison

During the 1960's, our business community saw a rash of conglomerations in which huge corporate monoliths swallowed up smaller companies and tried to digest them. As part of this movement, many centralized computer systems were installed with an aim toward gathering up the reins of management, and thus allowing the conglomerate's directors to run the whole show. If you were an analyst on one of these large Management Information System (MIS) projects, you got to see the user-analyst relationship at its very worst. Users were dead set against their functions being conglomerated, and of course that's just what the MIS systems were trying to do. The philosophy of the 60's was that an adversary relationship between the analyst and the user could be very productive, that analysts could go in, as the representatives of upper management, and force the users to participate and comply.

Of course the record of such projects was absolutely dismal. I know of no conglomerate that made significant gains in centralization through a large Management Information System. The projects were often complete routs. Many conglomerates are now spinning off their acquisitions and finding it rather simple to do so because so little true conglomeration was ever achieved. Due to the experience of the 60's, the term Management Information System, even today, is likely to provoke stifled giggles in a group of computer people.

The lesson of the 60's is that no system is going to succeed without the active and willing participation of its users. Users have to be made aware of how the system will work and how they will make use of it. They have to be sold on the system. Their expertise in the business area must be made a key ingredient to system development. They must be kept aware of progress, and channels must be kept open for them to correct and tune system goals during development. All of this is the responsibility of the analyst. He is the users' teacher, translator, and advisor. This intermediary function is the most essential of all the analyst's activities.

1.1.3 Specification

The analyst is the middleman between the user, who decides what has to be done, and the development team, which does it. He bridges this gap with a Target Document. The business of putting this document together and getting it accepted by all parties is specification. Since the Target Document is the analyst's principal output, specification is the most visible of his activities.

If you visit the Royal Naval Museum at Greenwich, England, you will see the results of some of the world's most successful specification efforts, the admiralty models. Before any ship of the line was constructed, a perfect scale model had to be built and approved. The long hours of detail work were more than repaid by the clear understandings that come from studying and handling the models.

The success of the specification process depends on the product, the Target Document in our case, being able to serve as a model of the new system. To the extent that it helps you visualize the new system, the Target Document is the system model.

1.1.4 Cost-Benefit Analysis

The study of relative cost and benefits of potential systems is the feedback mechanism used by an analyst to select an optimal target. While Structured Analysis does not entail new methods for conduct of this study, it nonetheless has an important effect. An accurate and meaningful system model helps the user and the analyst perfect their vision of the new system and refine their estimates of its costs and benefits.

1.1.5 Feasibility Analysis

It is pointless to specify a system which defies successful implementation. Feasibility analysis refers to the continual testing process the analyst must go through to be sure that the system he is specifying can be implemented within a set of given constraints. Feasibility analysis is more akin to design than to the other analysis phase activities, since it involves building tentative physical models of the system and evaluating them for ease of implementation. Again, Structured Analysis does not prescribe new procedures for this activity. But its modeling tools will have some positive effect.

1.1.6 Estimating

Since analysis deals so heavily with a system which exists only on paper, it involves a large amount of estimating. The analyst is forever being called upon to estimate cost or duration of future activities, CPU load factors, core and disk extents, manpower allocation . . . almost anything. I have never heard of a project's success being credited to the fine estimates an analyst made; but the converse is frequently true — poor estimates often lead to a project's downfall, and in such cases, the analyst usually receives full credit.

Estimating is rather different from the other required analysis skills:

- *Nobody is an expert estimator.* You can't even take a course in estimating, because nobody is willing to set himself up as enough of an authority on the subject to teach it.
- *We don't build our estimating skills, because we don't collect any data about our past results.* At the end of a project we rarely go

back and carry out a thorough postmortem to see how the project proceeded. How many times have you seen project performance statistics published and compared to the original estimates? In my experience, this is done only in the very rare instance of a project that finishes precisely on time and on budget. In most cases, the original schedule has long since vanished from the record and will never be seen again.

- *None of this matters as much as it ought to anyway*, since most things we call "estimates" in computer system projects are not estimates at all. When your manager asks you to come up with a schedule showing project completion no later than June 1 and using no more than six people, you're not doing any real estimating. You are simply dividing up the time as best you can among the phases. And he probably didn't estimate either; chances are his dates and manpower loading were derived from budgetary figures, which were themselves based upon nothing more than Wishful Thinking.

All these factors aside, estimating plays a key part in analysis. There are some estimating heuristics that are a by-product of Structured Analysis; these will be discussed in a subsequent chapter. The key word here is *heuristic*. A heuristic is a cheap trick that often works well but makes no guarantee. It is not an algorithm, a process that leads to a guaranteed result.

1.1.7 The Defensive Nature of Analysis

In addition to the analysis phase activities presented above, there are many others; the analyst is often a project utility infielder, called upon to perform any number of odd jobs. As the project wears on, his roles may change. But the major activities, and the ones that will concern us most in this book, are: user liaison, specification, cost-benefit and feasibility analysis, and estimating.

In setting about these activities, the analyst should be guided by a rule which seems to apply almost universally: *The overriding concern of analysis is not to achieve success, but to avoid failure.* Analysis is essentially a defensive business.

This melancholy observation stems from the fact that the great flaming failures of the past have inevitably been attributable to analysis phase flaws. When a system goes disastrously wrong, it is the analyst's fault. When a system succeeds, the credit must be apportioned among many participants, but failure (at least the most dramatic kind) belongs completely to the analyst. If you think of a system project that was a true rout — years late, or orders of magnitude over budget, or totally unacceptable to the user, or utterly impossible to maintain — it almost certainly was an analysis phase problem that did the system in.

Computer system analysis is like child-rearing; you can do grievous damage, but you cannot ensure success.

My reason for presenting this concept here is to establish the following context for the rest of the book: The principal goal of Structured Analysis is to minimize the probability of critical analysis phase error. The tools of Structured Analysis are defensive means to cope with the most critical risk areas of analysis.

1.2 Problems of analysis

Projects can go wrong at many different points: The fact that we spend so much time, energy, and money on maintenance is an indication of our failures as designers; the fact that we spend so much on debugging is an indictment of our module design and coding and testing methods. But analysis failures fall into an entirely different class. When the analysis goes wrong, we don't just spend more money to come up with a desired result — we spend *much* more money, and often don't come up with any result.

That being the case, you might expect management to be super-conservative about the analysis phase of a project, to invest much more in doing the job correctly and thus avoid whole hosts of headaches downstream. Unfortunately, it is not as simple as that. Analysis is plagued with problems that are not going to be solved simply by throwing money at them. You may have experienced this yourself if you ever participated in a project where too much time was allocated to the analysis phase. What tends to happen in such cases is that work proceeds in a normal fashion until the major products of analysis are completed. In the remaining time, the project team spins its wheels, agonizing over what more it could do to avoid later difficulties. When the time is finally up, the team breathes a great sigh of relief and hurries on to design. Somehow the extra time is just wasted — the main result of slowing down the analysis phase and doing everything with exaggerated care is that you just get terribly bored. Such projects are usually every bit as subject to failures of analysis as others.

I offer this list of the major problems of analysis:

1. communication problems
2. the changing nature of computer system requirements
3. the lack of tools
4. problems of the Target Document
5. work allocation problems
6. politics

Before looking at these problems in more detail, we should note that none of them will be *solved* by Structured Analysis or by any other approach to analysis. The best we can hope for is some better means to grapple with them.

1.2.1 Communication Problems

A long-unsolved problem of choreography is the development of a rigorous notation to describe dance. Merce Cunningham, commenting on past failures to come up with a useful notation, has observed that the motor centers of the brain are separated from the reading and writing centers. This physical separation in the brain causes communication difficulties.

Computer systems analysis is faced with this kind of difficulty. The business of specification is, for the most part, involved in describing procedure. Procedure, like dance, resists description. (It is far easier to demonstrate procedure than to describe it, but that won't do for our purposes.) Structured Analysis attempts to overcome this difficulty through the use of graphics. When you use a picture instead of text to communicate, you switch mental gears. Instead of using one of the brain's serial processors, its reading facility, you use a parallel processor.

All of this is a highfalutin way to present a "lowfalutin" and very old idea: A picture is worth a thousand words. The reason I present it at all is that analysts seem to need some remedial work on this concept. When given a choice (in writing a Target Document, for instance) between a picture and a thousand words, most analysts opt unfailingly for the thousand words.

Communication problems are exacerbated in our case by the lack of a common language between user and analyst. The things we analysts work with — specifications, data format descriptions, flowcharts, code, disk and core maps — are totally inappropriate for most users. The one aspect of the system the user is most comfortable talking about is the set of human procedures that are his interface to the system, typically something we don't get around to discussing in great detail with him until well after analysis, when the user manuals are being written.

Finally, our communication problem is complicated by the fact that what we're describing is usually a system that exists only in our minds. There is no model for it. In our attempts to flesh out a picture of the system, we are inclined to fill in the physical details (CRT screens, report formats, and so forth) much too early.

To sum it up, the factors contributing to the communication problems of analysis are

1. the natural difficulty of describing procedure
2. the inappropriateness of our method (narrative text)
3. the lack of a common language between analyst and user
4. the lack of any usable early model for the system

1.2.2 The Changing Nature of Requirements

I sometimes think managers are sent to a special school where they are taught to talk about "freezing the specification" at least once a day during the analysis phase. The idea of freezing the specification is a sublime fiction. Changes won't go away and they can't be ignored. If a project lasts two years, you ought to expect as many legitimate changes (occasioned by changes in the way business is done) to occur during the project as would occur in the first two years after cutover. In addition to changes of this kind, an equal number of changes may arise from the user's increased understanding of the system. This type of change results from early, inevitable communication failures, failures which have since been corrected.

When we freeze a Target Document, we try to hold off or ignore change. But the Target Document is only an approximation of the true project target; therefore, by holding off and ignoring change, we are trying to proceed toward a target *without benefit of any feedback.*

There are two reasons why managers want to freeze the Target Document. First, they want to have a stable target to work toward, and second, an enormous amount of effort is involved in updating a specification. The first reason is understandable, but the second is ridiculous. *It is unacceptable to write specifications in such a way that they can't be modified.* Ease of modification has to be a requirement of the Target Document.

This represents a change of ground rules for analysis. In the past, it was expected that the Target Document would be frozen. It was a positive advantage that the document was impossible to change since that helped overcome resistance to the freeze. It was considered normal for an analyst to hold off a change by explaining that implementing it in the Target Document would require retyping every page. I even had one analyst tell me that the system, once built, was going to be highly flexible, so that it would be easier to put the requested change into the system itself rather than to put it into the specification!

Figures collected by GTE, IBM, and TRW over a large sample of system changes, some of them incorporated immediately and others deferred, indicate that the difference in cost can be staggering. It can cost two orders of magnitude more to implement a change after cutover than it would have cost to implement it during the analysis phase. As a rule of thumb, you should count on a 2:1 cost differential to result from deferring change until a subsequent project phase.[1]

My conclusion from all of this is that we must change our methods; we must begin building Target Documents that are highly maintainable. In fact, maintainability of the Target Document is every bit as essential as maintainability of the eventual system.

---

[1]See Barry Boehm's article, "Software Engineering," published in the *IEEE Transactions on Computers*, December 1976, for a further discussion of this topic.

1.2.3 The Lack of Tools

Analysts work with their wits plus paper and pencil. That's about it. The fact that you are reading this book implies that you are looking for some tools to work with. For the moment, my point is that most analysts don't have any.

As an indication of this, consider your ability to evaluate the products of each project phase. You would have little difficulty evaluating a piece of code: If it were highly readable, well submodularized, well commented, conformed to generally accepted programming practice, had no GOTO's, ALTER's, or other forms of pathology — you would probably be willing to call it a good piece of code. Evaluating a design is more difficult, and you would be somewhat less sure of your judgment. But suppose you were asked to evaluate a Target Document. Far from being able to judge its quality, you would probably be hard pressed to say whether it qualified as a Target Document at all. Our inability to evaluate any but the most incompetent efforts is a sign of the lack of analysis phase tools.

1.2.4 Problems of the Target Document

Obviously the larger the system, the more complex the analysis. There is little we can do to limit the size of a system; there are, however, intelligent and unintelligent ways to deal with size. An intelligent way to deal with size is to *partition.* That is exactly what designers do with a system that is too big to deal with conveniently — they break it down into component pieces (modules). Exactly the same approach is called for in analysis.

The main thing we have to partition is the Target Document. We have to stop writing Victorian novel specifications, enormous documents that can only be read from start to finish. Instead, we have to learn to develop dozens or even hundreds of "mini-specifications." And we have to organize them in such a way that the pieces can be dealt with selectively.

Besides its unwieldy size, the classical Target Document is subject to further problems:

- It is excessively redundant.
- It is excessively wordy.
- It is excessively physical.
- It is tedious to read and unbearable to write.

1.2.5 Work Allocation

Adding manpower to an analysis team is even more complicated than beefing up the implementation team. The more successful classical analyses are done by very small teams, often only one person. On rush projects, the analysis phase is sometimes shortchanged since people assume it will take forever, and there is no convenient way to divide it up.

I think it obvious that this, again, is a partitioning problem. Our failure to come up with an early partitioning of the subject matter (system or business area) means that we have no way to divide up the rest of the work.

1.2.6 Politics

Of course, analysis is an intensely political subject. Sometimes the analyst's political situation is complicated by communication failures or inadequacies of his methods. That kind of problem can be dealt with positively — the tools of Structured Analysis, in particular, will help.

But most political problems do not lend themselves to simple solutions. The underlying cause of political difficulty is usually the changing distribution of power and autonomy that accompanies the introduction of a new system. No new analysis procedures are going to make such an impending change less frightening.

Political problems aren't going to go away and they won't be "solved." The most we can hope for is to limit the effect of disruption due to politics. Structured Analysis approaches this objective by making analysis procedures more formal. To the extent that each of the analyst's tasks is clearly (and publicly) defined, and has clearly stated deliverables, the analyst can expect less political impact from them. Users understand the limited nature of his investigations and are less inclined to overreact. The analyst becomes less of a threat.

1.3 The user-analyst relationship

Since Structured Analysis introduces some changes into the user-analyst relationship, I think it is important to begin by examining this relationship in the classical environment. We need to look at the user's role, the analyst's role, and the division of responsibility between them.

1.3.1 What Is a User?

First of all, there is rarely just one user. In fact, the term "user" refers to at least three rather different roles:

- *The hands-on user*, the operator of the system. Taking an on-line banking system as an example, the hands-on users might include tellers and platform officers.
- *The responsible user*, the one who has direct business responsibility for the procedures being automated by the system. In the banking example, this might be the branch manager.
- *The system owner*, usually upper management. In the banking example, this might be the Vice President of Banking Operations.

Sometimes these roles are combined, but most often they involve distinctly different people. When multiple organizations are involved, you can expect the total number of users to be as much as three times the number of organizations.

The analyst must be responsible for communication with *all* of the users. I am continually amazed at how many development teams jeopardize their chances of success by failing to talk to one or more of their users. Often this takes the form of some person or organization being appointed "User Representative." This is done to spare the user the bother of the early system negotiations, and to spare the development team the bother of dealing with users. User Representatives would be fine if they also had authority to accept the system. Usually they do not. When it comes to acceptance, they step aside and let the real user come forward. When this happens, nobody has been spared any bother.

1.3.2 What Is an Analyst?

The analyst is the principal link between the user area and the implementation effort. He has to communicate the requirements to the implementors, and the details of how requirements are being satisfied back to the users. He may participate in the actual determination of what gets done: It is often the analyst who supplies the act of imagination that melds together applications and present-day technology. And, he may participate in the implementation. In doing this, he is assuming the role that an architect takes in guiding the construction of his building.

While the details may vary from one organization to the next, most analysts are required to be

- at ease with EDP concepts
- at ease with concepts particular to the business area
- able to communicate such concepts

1.3.3 Division of Responsibility Between Analyst and User

There is something terribly wrong with a user-analyst relationship in which the user specifies such physical details as hardware vendor, software vendor, programming language, and standards. Equally upsetting is the user who relies upon the analyst to decide how business ought to be conducted. What is the line that separates analyst functions from user functions?

I believe the analyst and the user ought to try to communicate across a "logical-physical" boundary that exists in any computer system project. Logical considerations include answers to the question, *What needs to be accomplished?* These fall naturally into the domain of the user. Physical considerations include answers to the question, *How shall we accomplish these things?* These are in the domain of the analyst.

1.4 What is Structured Analysis?

So far, most of what we have been discussing has been the classical analysis phase, its problems and failings. How is Structured Analysis different? To answer that question, we must consider

- New goals for analysis. While we're changing our methods, what new analysis phase requirements shall we consider?
- Structured tools for analysis. What is available and what can be adapted?

1.4.1 New Goals for Analysis

Looking back over the recognized problems and failings of the analysis phase, I suggest we need to make the following additions to our set of analysis phase goals:

- The products of analysis must be highly maintainable. This applies particularly to the Target Document.
- Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out.
- Graphics have to be used wherever possible.
- We have to differentiate between logical and physical considerations, and allocate responsibility, based on this differentiation, between the analyst and the user.
- We have to build a logical system model so the user can gain familiarity with system characteristics before implementation.

1.4.2 Structured Tools for Analysis

At the very least, we require three types of new analysis phase tools:

- Something to help us partition our requirement and document that partitioning before specification. For this I propose we use a *Data Flow Diagram,* a network of interrelated processes. Data Flow Diagrams are discussed in Chapters 4 through 10.
- Some means of keeping track of and evaluating interfaces without becoming unduly physical. Whatever method we select, it has to be able to deal with an enormous flood of detail — the more we partition, the more interfaces we have to expect. For our interface tool I propose that we adopt a set of *Data Dictionary* conventions, tailored to the analysis phase. Data Dictionary is discussed in Chapters 11 through 14.

- New tools to describe logic and policy, something better than narrative text. For this I propose three possibilities: *Structured English, Decision Tables,* and *Decision Trees.* These topics are discussed in Chapters 15 through 17.

1.4.3 Structured Analysis — A Definition

Now that we have laid all the groundwork, it is easy to give a working definition of Structured Analysis:

Structured Analysis is the use of these tools:

Data Flow Diagrams
Data Dictionary
Structured English
Decision Tables
Decision Trees

to build a new kind of Target Document, the Structured Specification.

Although the building of the Structured Specification is the most important aspect of Structured Analysis, there are some minor extras:

- estimating heuristics
- methods to facilitate the transition from analysis to design
- aids for acceptance test generation
- walkthrough techniques

1.4.4 What Structured Analysis Is Not

Structured Analysis deals mostly with a subset of analysis. There are many legitimate aspects of analysis to which Structured Analysis does not directly apply. For the record, I have listed items of this type below:

- cost-benefit analysis
- feasibility analysis
- project management
- performance analysis
- conceptual thinking (Structured Analysis might help you communicate better with the user; but if the user is just plain wrong, that might not be of much long-term benefit.)
- equipment selection
- personnel considerations
- politics

My treatment of these subjects is limited to showing how they fit in with the modified methods of Structured Analysis.

3 THE TOOLS OF STRUCTURED ANALYSIS

The purpose of this chapter is to give you a look at each one of the tools of Structured Analysis at work. Once you have a good idea of what they are and how they fit together, we can go back and discuss the details.

3.1 A sample situation

The first example I have chosen is a real one, involving the workings of our own company, Yourdon inc. To enhance your understanding of what follows, you ought to be aware of these facts:

1. Yourdon is a medium-sized computer consulting and training company that teaches public and inhouse sessions in major cities in North America and occasionally elsewhere.
2. People register for seminars by mail and by phone. Each registration results in a confirmation letter and invoice being sent back to the registrant.
3. Payments come in by mail. Each payment has to be matched up to its associated invoice to credit accounts receivable.
4. There is a mechanism for people to cancel their registrations if they should have to.
5. Once you have taken one of the company's courses, or even expressed interest in one, your name is added to a data base of people to be pursued relentlessly forever after. This data base contains entries for tens of thousands of people in nearly as many organizations.
6. In addition to the normal sales prompting usage of the data base, it has to support inquiries such as
   - When is the next Structured Design Programming Workshop in the state of California?
   - Who else from my organization has attended the Structured Analysis seminar? How did they rate it?
   - Which instructor is giving the Houston Structured Design and Programming Workshop next month?

In early 1976, Yourdon began a project to install a set of automated management and operational aids on a PDP-11/45, running under the UNIX operating system. Development of the system — which is now operational — first called for a study of sales and accounting functions. The study made use of the tools and techniques of Structured Analysis. The following subsections present some partial and interim products of our analysis.

3.2 A Data Flow Diagram example

An early model of the operations of the company is presented in Fig. 9. It is in the form of a Logical Data Flow Diagram. Refer to that figure now, and we'll walk through one of its paths. The rest should be clear by inference.

Input to the portrayed area comes in the form of Transactions ("Trans" in the figure). These are of five types: Cancellations, Enrollments, Payments, Inquiries, plus those that do not qualify as any of these, and are thus considered Rejects. Although there are no people or locations or departments shown on this figure (it is logical, not physical), I will fill some of these in for you, just as I would for a user to help him relate back to the physical situation that he knows. The receptionist (a physical consideration) handles all incoming transactions, whether they come by phone or by mail. He performs the initial edit, shown as Process 1 in the figure. People who want to take a course in Unmitigated Freelance Speleology, for example, are told to look elsewhere. Incomplete or improperly specified enrollment requests and inquiries, etc., are sent back to the originator with a note. Only clean transactions that fall into the four acceptable categories are passed on.

Enrollments go next to the registrar. His function (Process 2) is to use the information on the enrollment form to update three files: the People File, the Seminar File, and the Payments File. He then fills out an enrollment chit and passes it on to the accounting department. In our figure, the enrollment chit is called "E-Data," and the accounting process that receives it is Process 6.

Information on the chit is now transformed into an invoice. This process is partially automated, by the way — a ledger machine is used — but that information is not shown on a logical Data Flow Diagram.

The invoice passes on to the confirmation process (which happens to be done by the receptionist in this case). This task (Process 7) involves combining the invoice with a customized form letter, to be sent out together as a confirmation. The confirmation goes back to the customer.

3.2.1 Some Data Flow Diagram Conventions

If you have followed the narrative so far, you have already picked up the major Data Flow Diagram conventions:

- *The Data Flow Diagram shows flow of data, not of control.* This is the difference between Data Flow Diagrams and flowcharts. The Data Flow Diagram portrays a situation from the point of view of the data, while a flowchart portrays it from the point of view of those who act upon the data. For this reason, you almost never see a loop in a Data Flow Diagram. A loop is something that the data are unaware of; each datum typically goes through it once, and so from its point of view it is not a loop at all. Loops and decisions are control considerations and do not appear in Data Flow Diagrams.

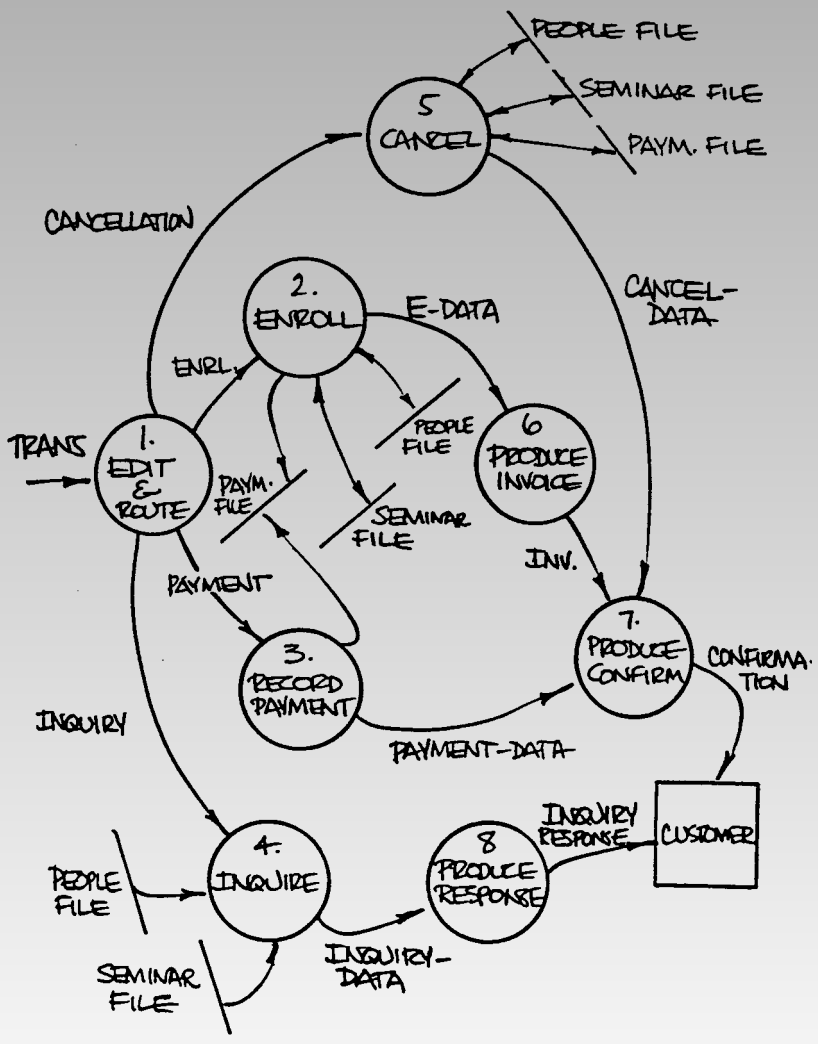
PEOPLE FILE
SEMINAR FILE
PAYM. FILE
5
CANCEL
CANCELLATION
CANCEL-
DATA
2.
ENROLL
E-DATA
ENRL
TRANS
1.
EDIT
&
ROUTE
PAYM.
FILE
PEOPLE
FILE
SEMINAR
FILE
6
PRODUCE
INVOICE
INV.
PAYMENT
3.
RECORD
PAYMENT
7.
PRODUCE
CONFIRM
CONFIRMA-
TION
INQUIRY
PAYMENT-DATA
INQUIRY
RESPONSE
CUSTOMER
8
PRODUCE
RESPONSE
4.
INQUIRE
PEOPLE
FILE
SEMINAR
FILE
INQUIRY-
DATA

Figure 9

- *Four notational symbols are used.* These are:
  - The named vector (called a data flow), which portrays a data path.
  - The bubble (called a process), which portrays transformation of data.
  - The straight line, which portrays a file or data base.
  - The box (called a source or sink), which portrays a net originator or receiver of data — typically a person or an organization outside the domain of our study.

Since no control is shown, you can't tell from looking at a Data Flow Diagram which path will be followed. The Data Flow Diagram shows only the set of possible paths. Similarly, you can't tell what initiates a given process. You cannot assume, for instance, that Process 6 is started by the arrival of an E-Data — in fact, that's not how it works at all. E-Data's accumulate until a certain day of the week arrives, and then invoices all go out in a group. So the data flow E-Data indicates the data path, but not the prompt. The prompting information does not appear on a Data Flow Diagram.

3.2.2 An Important Advantage of the Data Flow Diagram

Suppose you were walking through Fig. 9 with your user and he made the comment: "That's all very fine, but in addition to seminars, this company also sells books. I don't see the books operation anywhere."

"Don't worry, Mr. User," you reply, "the book operation is fully covered here," (now you are thinking furiously where to stick it) "here in Process . . . um . . . Process Number 3. Yes, definitely 3. It's part of recording payments, only you have to look into the details to see that."

Analysts are always good at thinking on their feet, but in this case, the effort is futile. The book operation has quite simply been *left out* of Fig. 9 — it's wrong. No amount of thinking on your feet can cover up this failing. No books flow in or out, no inventory information is available, no reorder data flows are shown. Process 3 simply doesn't have access to the information it needs to carry out books functions. Neither do any of the others.

Your only option at this point is to admit the figure is wrong and fix it. While this might be galling when it happens, in the long run you are way ahead — making a similar change later on to the hard code would cost you considerably more grief.

I have seen this happen so many times: an analyst caught flat-footed with an incorrect Data Flow Diagram, trying to weasel his way out, but eventually being forced to admit that it is wrong and having to fix it. I conclude that it is a natural characteristic of the tool:

> *When a Data Flow Diagram is wrong, it is glaringly, demonstrably, indefensibly wrong.*

This seems to me to be an enormous advantage of using Data Flow Diagrams.

3.2.3 What Have We Accomplished With a Data Flow Diagram?

The Data Flow Diagram is documentation of a situation from the point of view of the data. This turns out to be a more useful viewpoint than that of any of the people or systems that process the data, because the data itself sees the big picture. So the first thing we have accomplished with the Data Flow Diagram is to come up with a meaningful portrayal of a system or a part of a system.

The Data Flow Diagram can also be used as a model of a real situation. You can try things out on it conveniently and get a good idea of how the real system will react when it is finally built.

Both the conceptual documentation and the modeling are valuable results of our Data Flow Diagramming effort. But something else, perhaps more important, has come about as a virtually free by-product of the effort: The Data Flow Diagram gives us a highly useful *partitioning* of a system. Fig. 9 shows an unhandily large operation conveniently broken down into eight pieces. It also shows all the interfaces among those eight pieces. (If any interface is left out, the diagram is simply wrong and has to be fixed.)

Notice that the use of a Data Flow Diagram causes us to go about our partitioning in a rather oblique way. If what we wanted to do was break things down, why didn't we just do that? Why didn't we concentrate on functions and subfunctions and just accomplish a brute-force partitioning? The reason for this is that a brute-force partitioning is too difficult. It is too difficult to say with any assurance that some task or group of tasks constitutes a "function." In fact, I'll bet you can't even define the word function except in a purely mathematical sense. Your dictionary won't do much better — it will give a long-winded definition that boils down to saying a function is a bunch of stuff to be done. The concept of function is just too imprecise for our purposes.

The oblique approach of partitioning by Data Flow Diagram gives us a "functional" partitioning, where this very special-purpose definition of the word functional applies:

> A partitioning may be considered *functional* when the interfaces among the pieces are minimized.

This kind of partitioning is ideal for our purposes.

3.3 A Data Dictionary example

Refer back to Fig. 9 for a moment. What is the interface between Process 3 and Process 7? As long as all that specifies the interface is the weak name "Payment-Data," we don't have a specification at all. "Payment-Data" could mean anything. We must state precisely what me mean by the data flow bearing that name in order for our Structured Specification to be anything more than a hazy sketch of the system. It is in the Data Dictionary that we state precisely what each of our data flows is made up of.

An entry from the sample project Data Dictionary might look like this:

```
Payment-Data      =  Customer-Name +
                     Customer-Address +
                     Invoice-Number +
                     Amount-of-Payment
```

In other words, the data flow called "Payment-Data" consists precisely of the items Customer-Name, Customer-Address, Invoice-Number, and Amount-of-Payment, concatenated together. They must appear in that order, and they must all be present. No other kind of data flow could qualify as a Payment-Data, even though the name might be applicable.

You may have to make several queries to the Data Dictionary in order to understand a term completely enough for your needs. (This also happens with conventional dictionaries — you might look up the term perspicacious, and find that it means sagacious; then you have to look up sagacious.) In the case of the example above, you may have to look further in the Data Dictionary to see exactly what an Invoice-Number is:

```
Invoice-Number    =  State-Code +
                     Customer-Account-Number +
                     Salesman-ID +
                     Sequential-Invoice-Count
```

Just as the Data Flow Diagram effects a partitioning of the area of our study, the Data Dictionary effects a top-down partitioning of our data. At the highest levels, data flows are defined as being made up of subordinate elements. Then the subordinate elements (also data flows) are themselves defined in terms of still more detailed subordinates.

Before our Structured Specification is complete, there will have to be a Data Dictionary entry for every single data flow on our Data Flow Diagram, and for all the subordinates used to define them. In the same fashion, we can use Data Dictionary entries to define our files.

3.4 A Structured English example

Partitioning is a great aid to specification, but you can't specify by partitioning alone. At some point you have to stop breaking things down into finer and finer pieces, and actually document the makeup of the pieces. In the terms of our Structured Specification, we have to state what it takes to do each of the data transformations indicated by a bubble on our Data Flow Diagram.

There are many ways we could go about this. Narrative text is certainly the most familiar of these. To the extent that we have partitioned sufficiently before beginning to specify, we may be spared the major difficulties of narrative description. However, we can do even better.

A tool that is becoming more and more common for process description is Structured English. Presented below is a Structured English example of a user's invoice handling policy from the sample analysis. It appears without clarification; if clarification is needed, it has failed in its intended purpose.

==========

POLICY FOR INVOICE PROCESSING

```
If the amount of the invoice exceeds $500,
     If the account has any invoice more than 60 days overdue,
          hold the confirmation pending resolution of the debt.
     Else (account is in good standing),
          issue confirmation and invoice.
Else (invoice $500 or less),
     If the account has any invoice more than 60 days overdue,
          issue confirmation, invoice and write message on the
          credit action report.
     Else (account is in good standing),
          issue confirmation and invoice.
```

==========

3.5 A Decision Table example

The same policy might be described as well by a Decision Table:

	RULES			
CONDITIONS	1	2	3	4
1. Invoice > $500	Y	N	Y	N
2. Account over-due by 60+ days	Y	Y	N	N
ACTIONS				
1. Issue Confirmation	N	Y	Y	Y
2. Issue Invoice	N	Y	Y	Y
3. Msg to C.A.R.	N	Y	N	N

3.6 A Decision Tree example

As a third alternative, you might describe the same policy with a Decision Tree. I have included the equivalent Decision Tree as Fig. 10.

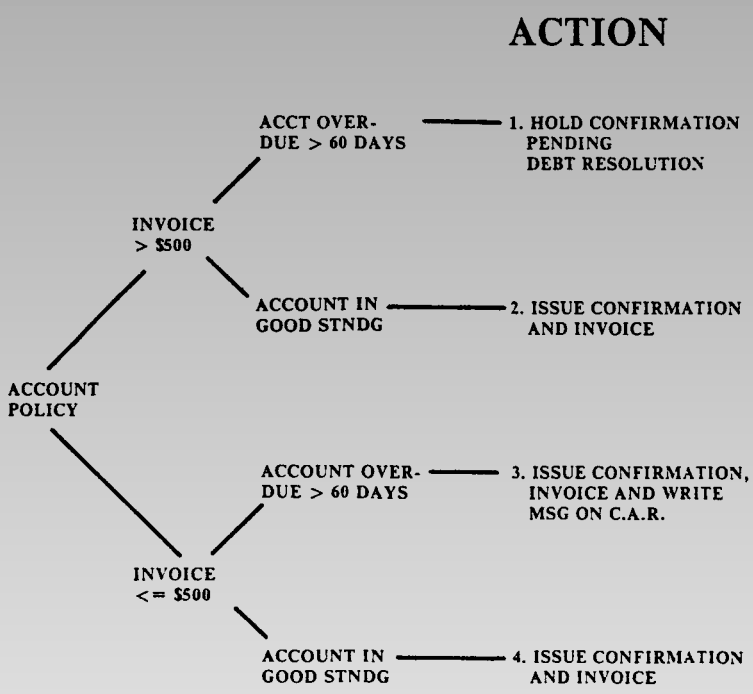
ACTION
ACCT OVER-
DUE > 60 DAYS
1. HOLD CONFIRMATION
PENDING
DEBT RESOLUTION
INVOICE
> $500
ACCOUNT IN
GOOD STNDG
2. ISSUE CONFIRMATION
AND INVOICE
ACCOUNT
POLICY
ACCOUNT OVER-
DUE > 60 DAYS
3. ISSUE CONFIRMATION,
INVOICE AND WRITE
MSG ON C.A.R.
INVOICE
<= $500
ACCOUNT IN
GOOD STNDG
4. ISSUE CONFIRMATION
AND INVOICE

Figure 10