Chapter 5: *The Project Life Cycle*

"All human error is impatience, a premature renunciation of method, a delusive pinning down of a delusion."

— Franz Kafka
Letters

In this chapter, you will learn:

1. The concept of a project life cycle;
2. The characteristics of the classical project life cycle;
3. The difference between classical and semistructured projects;
4. The components of the structured life cycle; and
5. The difference between radical and conservative life cycles.

To be an effective systems analyst, we need more than just modeling tools; we need *methods*. In the systems development profession, the terms method, methodology, project life cycle, and systems development life cycle are used almost interchangeably. In Part III, we will look at the detailed methods for doing systems analysis, within the broader context of a method — known as a structured project life cycle — for carrying out the overall development of a new system.

Before introducing the structured project life cycle, it is important to examine the classical project life cycle discussed in many textbooks and used in many systems development organizations today, primarily to identify its limitations and weaknesses. This examination will be followed by a brief discussion of the semistructured life project life cycle: a project life cycle that includes some, but not all, of the elements of modern systems development. Next, we introduce the structured project life cycle, presenting an overview to show the major activities and how they fit together. Finally, we will examine briefly the *iterative*, or *prototyping*, life cycle introduced in the mid-1980s by Bernard Boar and James Martin, and now popularized by software engineers such as James Highsmith and Kent Beck (see (Highsmith, 2000) and (Beck, 2000)).

We will also explore the concept of iterative or top-down *development*. In particular, we will introduce the notion of *radical* top-down development and *conservative* top-down development. Depending on the nature of a systems development project, there may be valid reasons for adopting one approach rather than the other; indeed, some projects may call for a combination of the two.

5.1 The concept of a project life cycle

As you might expect, small IT organizations tend to be relatively informal: systems development projects are begun as the result of a verbal discussion between the user and the project manager (who may also be the systems analyst, programmer, computer operator, and janitor!), and the project proceeds from systems analysis through design and implementation without much fuss.

In the larger organizations, however, things are done on a much more formal basis. The various communications between users, management, and the project team tend to be documented in writing, and everyone understands that the project will go through several phases before it is complete. Even so, it is surprising to see the major differences between the way two project managers in the same organization will conduct their respective projects. Indeed, it is often left to the discretion of the individual project manager to determine what phases and activities her or his project will consist of and how these phases will be conducted. **[1]**

Recently, though, the approach taken to systems development has begun to change. More and more large and small organizations are adopting a single, uniform project life cycle — sometimes known as a project plan or systems development methodology or, simply, "the way we do things around here." Usually contained in a notebook as ponderous as the standards manual that sits (unread) on every analyst's and programmer's desk, the documented project life cycle provides a common way for everyone in the systems development organization to go about the business of developing a computer system.

The approach may be home-grown or, alternatively, the systems development organization may decide to purchase a project management package and then tailor it to company needs. **[2]** It seems apparent that, aside from providing employment for the people who create project life cycle manuals (and for those who write textbooks about them!), the project methodology is desirable. What then is the purpose of having a project life cycle? There are three primary objectives:

1. To define the activities to be carried out in a systems development project.
2. To introduce consistency among many systems development projects in the same organization.
3. To provide checkpoints for management control for go/no-go decisions.

The first objective is particularly important in a large organization in which new people are constantly entering the ranks of project management. The fledgling project manager may overlook or underestimate the significance of important project phases if he or she follows only intuition. Indeed, it can happen that junior programmers and systems analysts may not understand where and how their efforts fit into the overall project unless they are given a proper description of all the phases of the project.

The second objective is also important in a large organization. For higher levels of management, it can be extremely disconcerting to oversee a hundred different projects, each of which is being carried out in a different way. For example, if project A defines the systems analysis activity differently than does project B, and project B doesn't include a design phase, how is the second- or third-level manager to know which project is in trouble and which is proceeding on schedule? **[3]**

The third objective of a standard project life cycle is associated with management's need to control a project. On trivial projects, the sole checkpoint is likely to be the end of the project: Was it finished on time and within the specified budget? (Or even more simply: was it finished at all?) And did it accomplish the user's requirements? But for larger projects, management should have a number of intermediate checkpoints during the project, which provide it with opportunities to determine whether the project is behind schedule, and whether additional resources need to be procured. In addition, an intelligent user will also want checkpoints at several stages in the project so that he can determine whether he wants to continue funding it! **[4]**

Having said all this, let me emphasize that the project life cycle definitely is not in charge of the project. It will not relieve the project manager of the difficult responsibility of making decisions, weighing alternatives, fighting political battles, negotiating with recalcitrant users, boosting the morale of dejected programmers, or any of the other project-related trials and tribulations. The project manager still has to manage, in every sense of the word. The only help that the project life cycle can provide is that it can organize the manager's activities, making it more likely that the right problems will be addressed at the right time.

5.2 The classical project life cycle

The kind of project life cycle used in many organizations today differs from the one to which we'll be devoting most of our attention in Part III. The classical, or conventional, project life cycle is shown in Figure 5.1. Every project goes through some kind of systems analysis, design, and implementation, even if it's not done in exactly the way shown in the diagram. The project life cycle used in your organization, for example, might differ from the one shown in Figure 5.1 in one or all of the following ways:

* The survey phase and the analysis phase may be lumped together into a single phase (this is especially common in organizations in which anything the user wants is deemed at the outset to be feasible).
* There may not be a phase called hardware study if it can be taken for granted that any new system can be implemented on an existing computer without causing any major operational impact.
* The preliminary design and detail design phases may be lumped together in a single phase simply called design.
* Several of the testing phases may be grouped together into a single phase; indeed, they may even be included with coding.

Thus, an individual organization's project life cycle may have five phases, or seven phases, or twelve phases, but it is still likely to be of the classical variety.

What is it that really characterizes a project life cycle as being classical? Two features stand out: a strong tendency toward bottom-up implementation of the system, and an insistence on linear, sequential progression from one phase to the next.

5.2.1 Bottom-up implementation

The use of bottom-up implementation is one of the major weaknesses in the classical project life cycle. As you can see from Figure 5.1(a), the programmers are expected to carry out all their module testing (sometimes known as "component testing") first, then subsystem testing, and finally system testing. This approach is also known in the computer industry as the "waterfall life cycle," based on a diagram introduced in (Royce, 1970), and subsequently popularized by Barry Boehm in (Boehm, 1981). It is shown in Figure 5.1(b).

It's not clear where this approach originally came from, but it may have been borrowed from assembly-line manufacturing industries. The bottom-up implementation approach is a good one for assembling automobiles on an assembly line, *but only after the prototype model has been thoroughly debugged!* **[5]** Unfortunately, many system development organizations are still producing one-of-a-kind systems, for which the bottom-up approach has a number of serious difficulties:

* Nothing is done until it's *all* done. Thus, if the project gets behind schedule and the deadline falls right in the middle of system testing, there will be nothing to show the user except an enormous pile of program listings — which, taken in their entirety, do nothing of any value for the user!

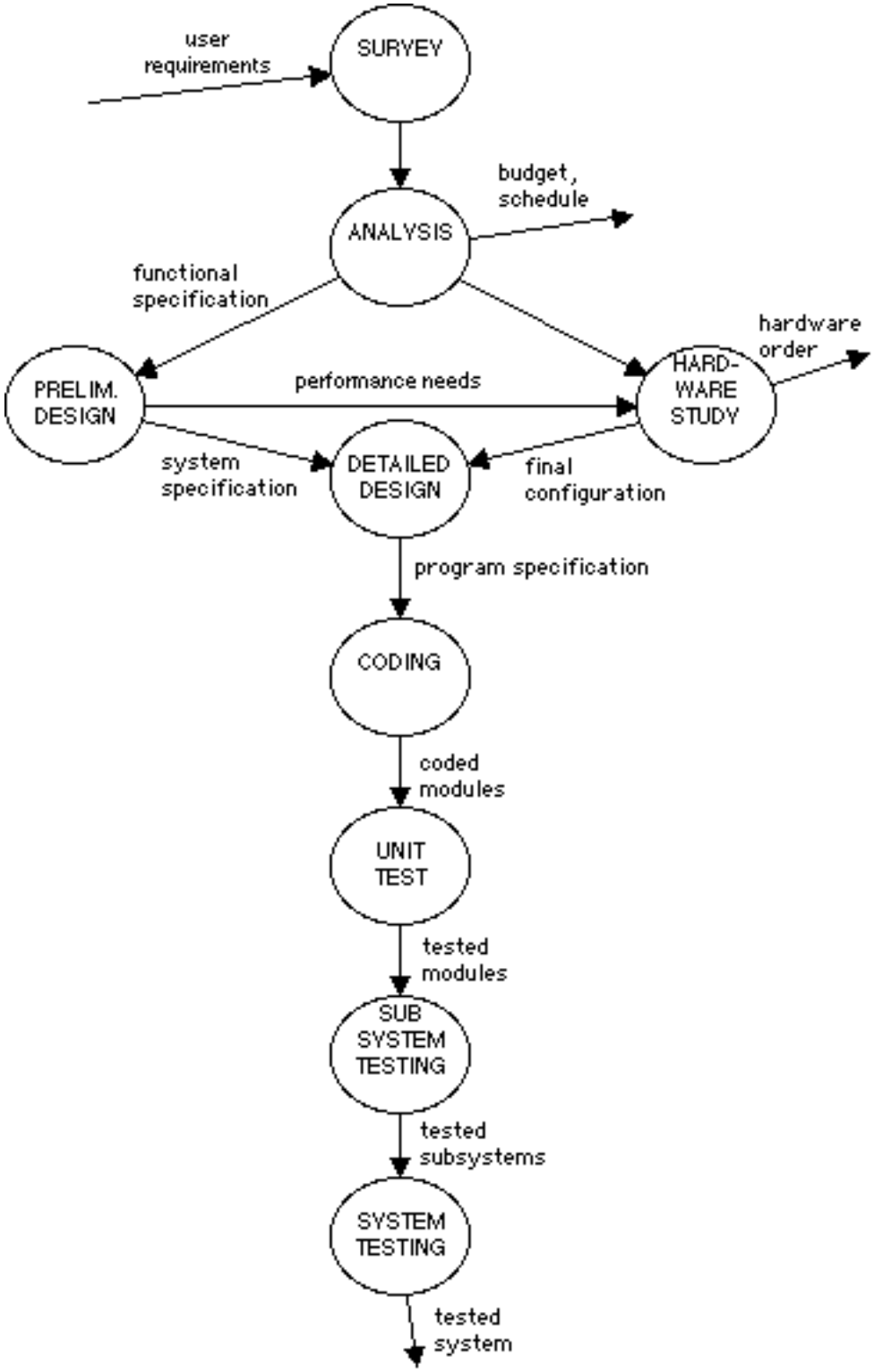

Figure 5.1(a): The classical project life cycle

* The most trivial bugs are found at the beginning of the testing period, and the most serious bugs are found last. This is almost a tautology: Module testing uncovers relatively simple logic errors inside individual modules; system testing, on the other hand, uncovers major interface errors between subsystems. The point is that major interface errors are *not* what the programmer wants to find at the end of a development project; such bugs can lead to the recoding of large numbers of modules, and can have a devastating impact on the schedule, right at a time when everyone is likely to be somewhat tired and cranky from having worked so hard for so many months.
* Debugging tends to be extremely difficult during the final stages of system testing. Note that we distinguish here between *testing* and *debugging*. Debugging is the black art of discovering *where* a bug is located (and the subsequent determination of how to fix the bug) after the process of testing has determined that there *is* a bug. When a bug is discovered during the system-testing phase of a bottom-up project, it's often extremely difficult to tell which module contains the bug; it could be in any one of the hundreds (or thousands) of modules that have been combined for the first time. Tracking down the bug is often like looking for a needle in a haystack.
* The requirement for computer test time usually rises exponentially during the final stages of testing. More specifically, the project manager often finds that she or he needs large contiguous chunks of computer time for system testing, perhaps 12 hours of uninterrupted computer time per day. Since such a large amount of computer time is often difficult to obtain **[6]**, the project often falls seriously behind schedule.

5.2.2 Sequential progression

The second major weakness with the classical project life cycle is its insistence that the phases proceed sequentially from one to the next. There is a natural, human tendency to want this to be so: We want to be able to say that we have *finished* the systems analysis phase and that we'll never have to worry about that phase again. Indeed, many organizations formalize this notion with a ritual known as freezing the specification or freezing the design document.

The only problem with this desire for orderly progression is that it's completely unrealistic! In particular, the sequential approach doesn't allow for real-world phenomena having to do with personnel, company politics, or economics. For example, the person doing the work, such as the systems analyst or designer, may have made a mistake, and may have produced a flawed product. Indeed, as human beings, we rarely do a complex job right the first time, but we are very good at making repeated improvements to an imperfect job. Or the person reviewing the work, in particular, the user who reviews the work of the systems analyst, may have made a mistake. Or perhaps the person carrying out the work associated with each phase may not have enough time to finish, but may be unwilling to admit that fact. This is a polite way of saying that, on most complex projects, systems analysis and design (and system testing, too) finish when someone decides that you have run out of time, not when you want those activities to finish!

Other problems are commonly associated with the sequential, classical project life cycle: During the several months (or years) that it takes to develop the system, the user may change his or her mind about what the system should do. During the period that it takes to develop the system, certain aspects of the user's environment may change (e.g., the economy, the competition, or the government regulations that affect the user's activities).

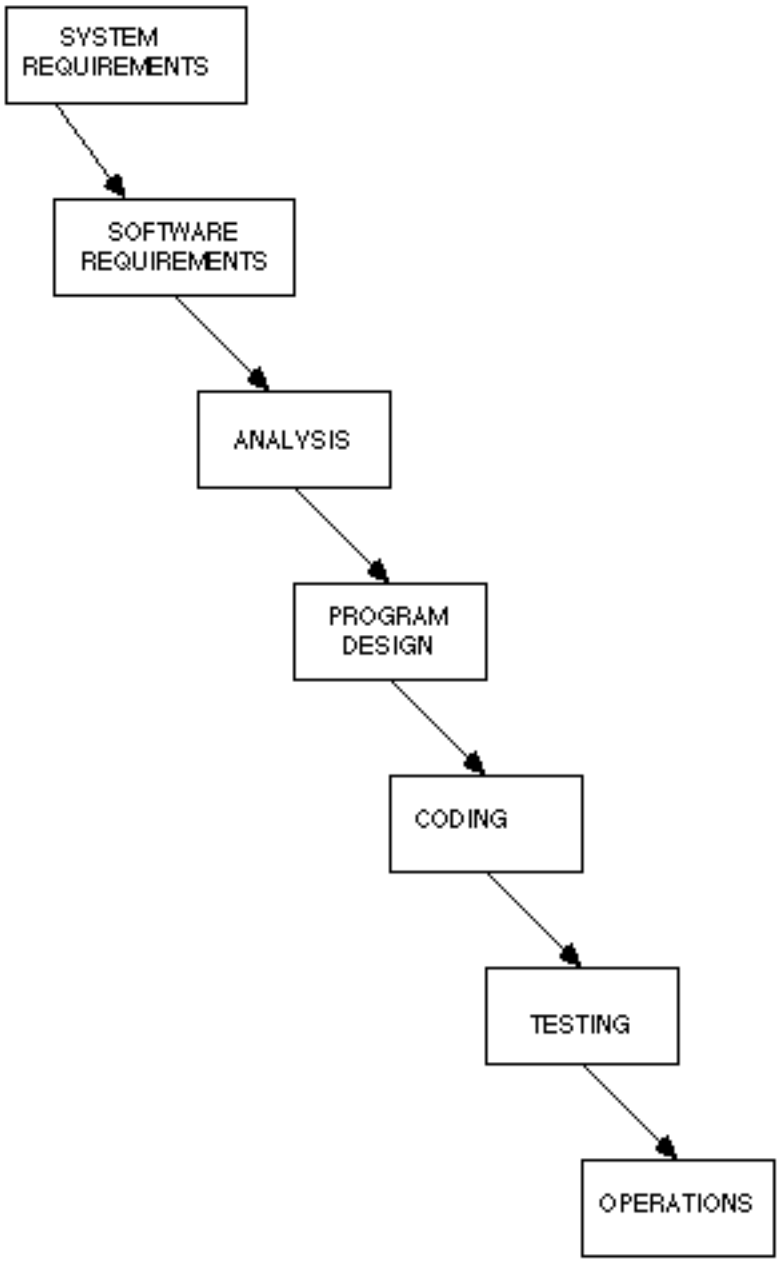

Figure 5.1(b): The waterfall model of systems development

An additional characteristic of the classical project life cycle is that it relies on outdated techniques. That is, it tends to make no use of structured design, structured programming, walkthroughs, or any of the other modern development techniques. **[7]** But because the classical project life *ignores* the existence of these techniques, there is nothing to prevent the project manager from using them. Unfortunately, many programmers, systems analysts, and first-level project leaders feel that the project life cycle is a statement of policy by top-level management; and if management doesn't say anything about the use of structured programming, then they, as mere project members and leaders, are not obliged to use nonclassical approaches.

5.3 The semistructured life cycle

The comments in the previous section make it seem as if most IT organizations are still living in the Dark Ages. Indeed, such a characterization is unfair: Not every organization uses the classical project life. Beginning in the early 1980s, there IT industry began recognizing that formal, disciplined techniques like structured design, structured programming, and top-down implementation had some significant benefits. This recognition has led to the semistructured project life cycle shown in Figure 5.2; it shows two obvious features not present in the classical approach:

1. The bottom-up sequence of coding, module testing, and system testing is replaced by top-down implementation, an approach where high-level modules are coded and tested first, followed by the lower-level, detailed modules. There is also a strong indication that structured programming is to be used as the method of actually coding the system.
2. Classical design is replaced by structured design, a formal systems design approach discussed in such books as (Yourdon and Constantine, 1989) and (Page-Jones, 1988).

Aside from these obvious differences, there are some subtle points about this modified life cycle. Consider, for example, that top-down implementation means that some coding and testing are taking place in parallel. That certainly represents a major departure from the sequential phases that we saw in the classical life cycle! In particular, it can mean *feedback* between the activity of coding and that of testing and debugging. When the programmer tests the top-level skeleton version of the system, he or she may be heard to mutter, "Jeez, I had no idea that the double-precision FRAMMIS instruction worked *that* way!" Naturally, you can be sure that subsequent use of the double-precision FRAMMIS instruction will be quite different.

Perhaps more important, the use of top-down implementation tempts the implementors (and the systems analysts, if they haven't abandoned the project by this time) to talk to the users *after* the specifications have been ceremoniously frozen. Thus, it is possible that the user will point out errors or misunderstandings in the specification; the user may even express a desire to *change* the specification, and if the conversation takes place directly between the user and the implementor, a change may actually be effected before the project manager knows what is happening. In short, top-down implementation often provides feedback between the implementation process and the analysis process — although this feedback exchange is not specifically shown on Figure 5.2, and although the user and the IT project manager might well deny that it is taking place!

There is one final point about the semistructured life cycle: A significant part of the work that takes place under the heading of "structured design" is actually a manual effort to fix up bad narrative specifications. You can see this by looking at Figure 5.3, which depicts the details of structured design. (Note that this figure consists of the details of process 3 in Figure 5.2.)

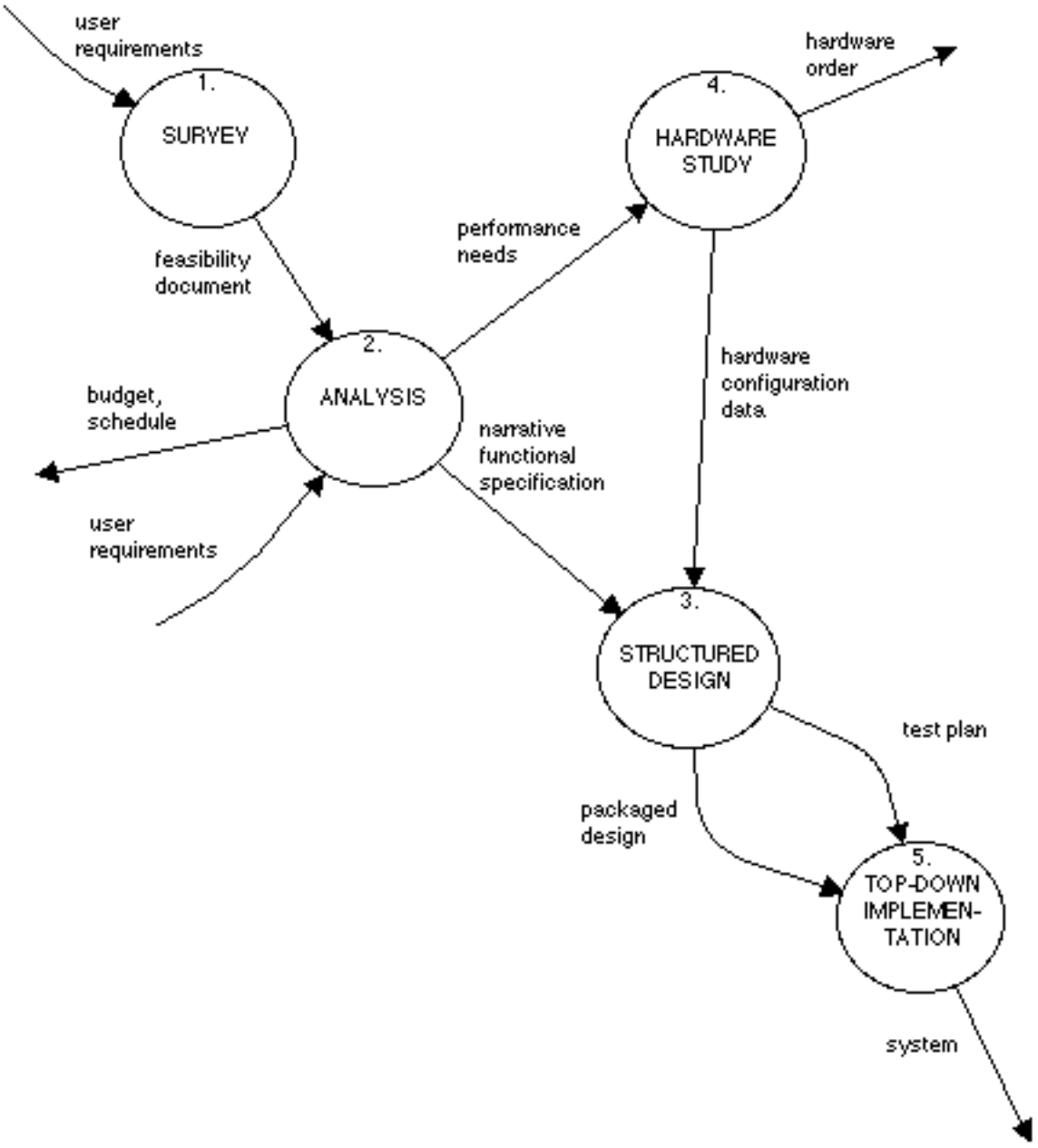

Figure 5.2: The semistructured project life cycle

In Figure 5.3, activity 3.1 (labeled CODIFY FUNCTIONAL SPECIFICATION) represents a task that designers have long had to do: translate a monolithic, ambiguous, redundant, narrative document into a useful, nonprocedural model to serve as the basis for deriving the hierarchy of modules that will implement the user's requirements. In other words, people practicing structured design have traditionally assumed that they would be given a classical specification; consequently, their first job, as they see it, is to transform that specification into a package of dataflow diagrams, data dictionaries, entity-relationship diagrams, and process specifications.

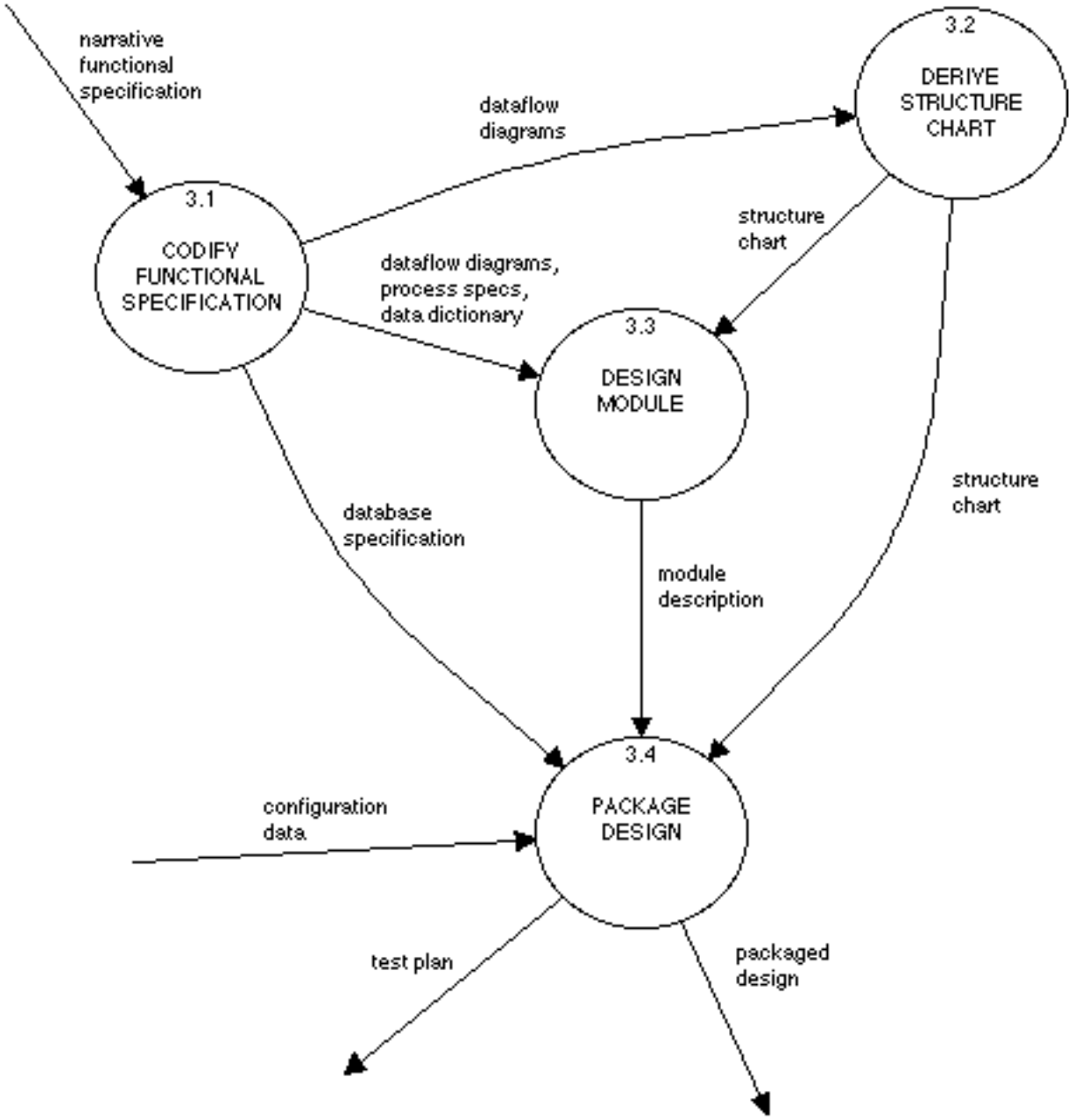

Figure 5.3: Details of the design activity

This is a more difficult job than you might imagine: Historically, it has been a task carried out in a vacuum. Designers generally had little contact with the systems analyst who wrote the long narrative specification, and they certainly had no contact with the user!

Obviously, such a situation is ripe for change. Introducing structured analysis — the kind of modern systems analysis approach presented in this book — into the picture, as well as expanding on the idea of feedback between one part of the project and another, creates an entirely different kind of project life cycle. This is the structured project life cycle, which we will discuss next.

5.4 The structured project life cycle

Now that we have seen the classical project life cycle and the semistructured project life cycle, we are ready to examine the structured life cycle; it is shown in Figure 5.4.

We will briefly examine the project life cycle's nine activities and three terminators, as shown in Figure 5.4. The terminators consist of users, managers, and operations personnel; as you recall, we discussed their roles in Chapter 3. These are individuals or groups of individuals who provide input to the project team and who are the ultimate recipients of the system. They interact with the nine activities that we have shown in Figure 5.4. Each of the activities is summarized in the following sections.

5.4.1 Activity 1: the survey

This activity is also known as the feasibility study or initial business study. Typically, it begins when a user requests that one or more portions of his or her business be automated. The major purposes of the survey activity are as follows:

* *Identify the responsible users and develop an initial "scope" of the system*. This may involve conducting a series of interviews to see which user(s) are involved in (or affected by) the proposed project and which are not. **[8]**
* It may also involve developing an initial context diagram — a simple dataflow diagram of the sort shown in Figure 4.2, in which the entire system is represented by a single process. **[9]**
* *Identify current deficiencies in the user's environment*. This will usually consist of a simple narrative list of functions that are missing or operating unacceptably in the current system. For example, this might include statements like the following:
* The hardware for the current system is unreliable, and the vendor has just gone bankrupt.
* The software for the current system is unmaintainable, and we can no longer hire maintenance programmers who are willing to maintain software in the programming language used to develop the current system.
* The response time for the current on-line order entry system is so bad that many customers hang up, in frustration, before they have entered their order.
* The current system is unable to produce the government reports required by last year's Tax Reform Act.
* The current system is unable to receive credit-limit reports from the accounting department and cannot produce reports of average order size for the marketing department.

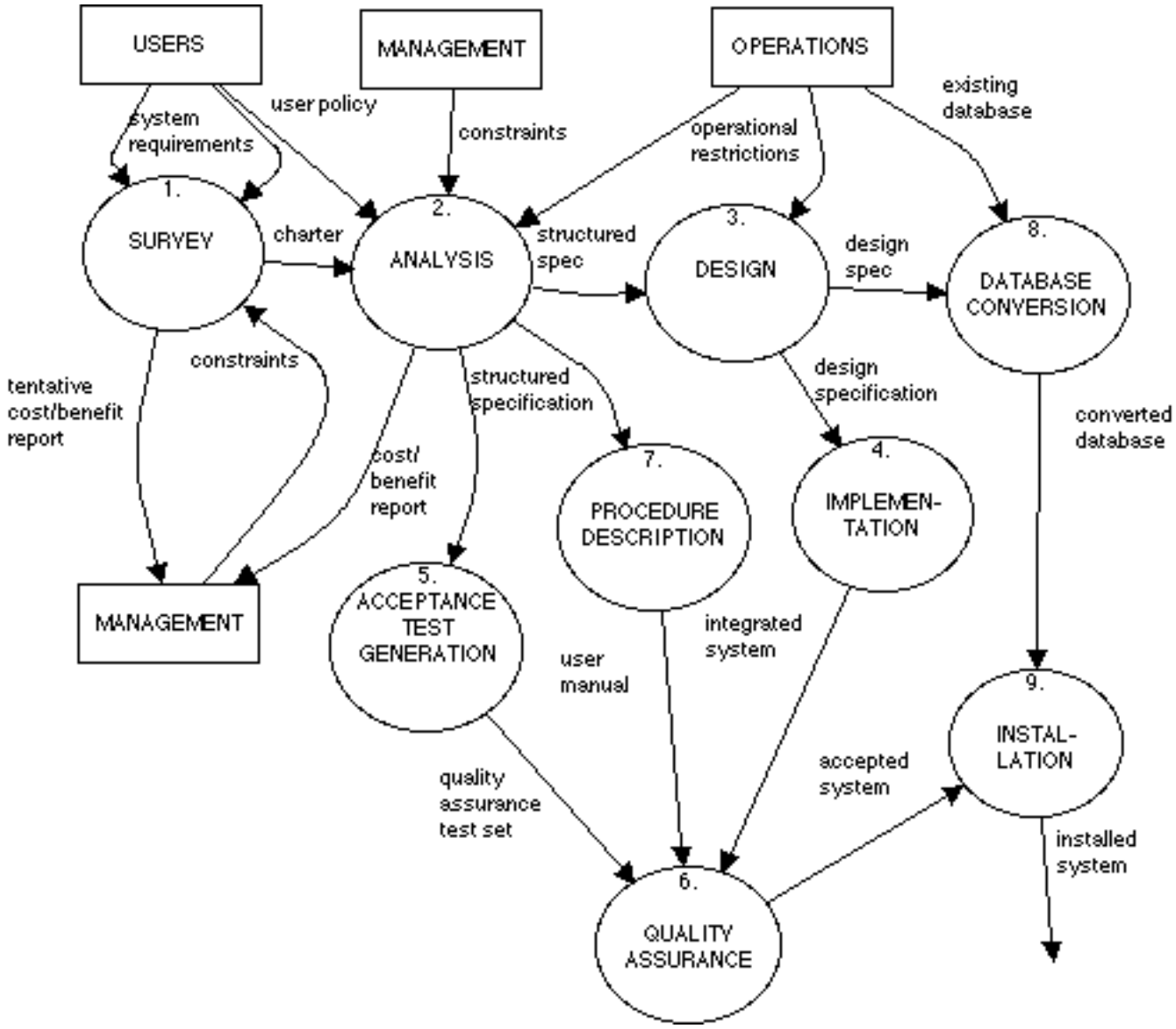

Figure 5.4: The structured project life cycle

* *Establish goals and objectives for a new system*. This may also be a simple narrative list consisting of existing functions that need to be re-implemented, new functions that need to be added, and performance criteria for the new system.

* *Determine whether it is feasible to automate the system and, if so, suggest some acceptable scenarios*. This will involve some very crude and approximate estimates of the schedule and cost to build a new system and the benefits to be derived **[10]**; it will also involve two or more scenarios (e.g., the mainframe scenario, the distributed processing scenario, etc.). Though management and the users will often want a precise, detailed estimate at this point, the systems analyst will be extremely lucky if she or he can estimate the time, resources, and costs to within ±50% at this early stage in the project.

* *Prepare a project charter that will be used to guide the remainder of the project*. The project charter will include all the information listed above, as well as identify the responsible project manager. It may also describe the details of the project life cycle that the rest of the project will follow.

The survey typically occupies only 5% to 10% of the time and resources of the entire project, and for small, simple projects it may not even be a formal activity. However, even though it may not consume much of the time or resources of the project, it is a critical activity: at the end of the survey, management may decide to cancel the project if it does not appear attractive from a cost-benefit point of view.

As a systems analyst, you may or may not be involved in the survey; the user, together with appropriate levels of management, may have done it before you even hear about the project. However, for large, complex projects, the survey involves enough detailed work that the user will often ask the systems analyst to be involved as early as possible.

We will not discuss the survey in any further detail in this book. If you become involved in this activity, you may find Appendices E and C useful. For additional details, consult (Dickinson, 1981), (Gore and Stubbe, 1983), and (Yourdon, 1988).

5.4.2 Activity 2: systems analysis

The primary purpose of the analysis activity is to transform its two major inputs, user policy and project charter, into a structured specification. This involves modeling the user's environment with dataflow diagrams, entity-relationship diagrams, state-transition diagrams, and the other tools presented in Chapter 4; these tools are covered in detail in Part II.

The step-by-step process of systems analysis (i.e., the subactivities of the analysis activity in Figure 5.4) is discussed in Part III. As we will see, it involves the development of an *environmental model* (discussed in Chapter 18) and the development of a *behavioral model* (discussed in Chapters 19 and 20). These two models combine to form the *essential model* (discussed in Chapter 17), which represents a formal description of what the new system must do, independent of the nature of the technology that will be used to implement those requirements.

In addition to the system model describing user requirements, a more accurate, detailed set of budgets and cost-benefit calculations is generally prepared at the end of the analysis Activity. This is discussed in more detail in Appendix C.

Obviously, as a systems analyst, this is where you will be spending the major part of your time. There is nothing more we need to say about the analysis activity at this point, since it is the subject of the rest of this book.

5.4.3 Activity 3: design

The activity of design is concerned with allocating portions of the specification (otherwise known as the essential model) to appropriate processors (CPUs and/or humans) and to appropriate tasks (or jobs, or partitions, etc.) within each processor. Within each task, the design activity is concerned with the development of an appropriate hierarchy of program modules and interfaces between those modules to implement the specification created in activity 2. In addition, the design activity is concerned with the transformation of entity-relationship data models into a database design; see (Inmon, 1988) for more details.

Part of this activity will be of interest to you as a systems analyst: the development of something called the *user implementation model*. This model describes those implementation issues that the user feels strongly enough about that he or she is not willing to leave them to the discretion of the systems designers and programmers. The primary issues that the user is typically concerned about is the specification of the human-machine boundary and the specification of the human interface. The human-machine boundary separates those parts of the essential model that are to be carried out by a person (as a manual activity) from the parts that are to be implemented on one or more computers. Similarly, the human interface is a description of the format and sequence of *inputs* provided by human users to the computer (e.g., screen designs and the on-line dialogue between the user and the computer), as well as the format and sequence of *outputs* provided by the computer to the user. The user implementation model is described in Chapter 21.

An introduction to the process of systems design may be found in Chapter 22. Additional material may be found in (Yourdon and Constantine, 1989), (Page-Jones, 1988), (Jackson, 1975), and others.

5.4.4 Activity 4: implementation

This activity includes both coding and the integration of modules into a progressively more complete skeleton of the ultimate system. Thus, Activity 4 includes both structured programming and top-down implementation.

As you can imagine, this is typically an activity that the systems analyst is not involved with, though there are some projects (and some organizations) where systems analysis, design, and implementation are done by the same person. This topic is discussed in more detail in Chapter 23.

5.4.5 Activity 5: acceptance test generation

The structured specification should contain all the information necessary to define an acceptable system from the user's point of view. Thus, once the specification has been generated, work can commence on the activity of generating a set of acceptance test cases from the structured specification.

Since the development of acceptance tests can take place in parallel with the activities of design and implementation, it is possible that you may be assigned to this task after you finish developing the essential model in Activity 2. Testing is discussed in more detail in Chapter 23.

5.4.6 Activity 6: quality assurance

Quality assurance is also known as final testing or acceptance testing. This activity requires, as its input, acceptance test data generated in Activity 5 and an integrated system produced by Activity 4.

The systems analyst could conceivably be involved in the quality assurance activity, but typically is not. One or more members of the user organization may take responsibility, or it may be carried out by an independent testing group or Quality Assurance Department. Consequently, we will not discuss the quality assurance function in any further detail.

Note, by the way, that some people refer to this activity as quality control rather than quality assurance. Regardless of the terminology, we need an activity to verify that the system exhibits the appropriate level of quality; this is what we have called quality assurance in this book. Note also that it is important to carry out quality assurance activities throughout *each* of the earlier activities to ensure that they have been performed at an appropriate level of quality. Thus, one would expect a QA activity to be performed throughout the analysis, design, and programming activity to ensure that the analyst is developing high-quality specifications, the designer is producing a high-quality design, and the programmer is writing high-quality code. The quality assurance activity identified here is merely the *final* test of the system's quality.

5.4.7 Activity 7: procedure description

Throughout this book, we concern ourselves with the development of an entire system — not just the automated portion, but also the portion to be carried out by people. Thus, one of the important activities to be performed is the generation of a formal description of those portions of the new system that will be manual, as well as a description of how the users actually will interact with the automated portion of the new system. The output of Activity 7 is a user's manual.

As you can imagine, this is also an activity in which you may become involved as a systems analyst. Though we do not discuss it further in this book, you may wish to consult books on technical writing for additional information on the writing of user manuals.

5.4.8 Activity 8: database conversion

In some projects, database conversion involved more work (and more strategic planning) than did the development of computer programs for the new system; in other cases, there might not have been any existing database to convert. In the general case, this activity requires, as input, the user's current database, as well as the design specification produced by Activity 3.

Depending on the nature of the project, you may or may not become involved in the database conversion activity as a systems analyst. However, we do not discuss this activity in any greater detail in this book.

5.4.9 Activity 9: installation

The final activity, of course, is installation; its inputs are the user's manual produced by Actvity 7, the converted database produced by Activity 8, and the accepted system produced by Activity 6. In some cases, however, installation may simply mean an overnight "cutover" to the new system, with no excitement or fanfare; in other cases, installation may be a gradual process, as one user group after another receives user's manuals, hardware, and training in the use of the new system and actually begin using it.

5.4.10 Summary of the structured project life cycle

It's important that you view Figure 5.4 for what it is: a dataflow diagram. It is not a flowchart; there is no implication that all of activity N must finish before activity N + 1 commences. On the contrary, the network of dataflows connecting activities strongly implies that several activities may be going on in parallel. It is because of this nonsequential aspect that we use the word activity in the structured project life cycle, rather than the more conventional word phase. The term phase has traditionally referred to a particular period of time in a project when one, and only one, activity was going on.

There is something else that must be emphasized about the use of a dataflow diagram to depict the project life cycle: A classical dataflow diagram, such as the one in Figure 5.4, does not explicitly show feedback, nor does it show control **[11]**. Virtually every one of the activities in Figure 5.4 can, and usually does, produce information that can provide suitable modifications to one or more of the preceding activities. Thus, the activity of design can produce information that may revise some of the cost-benefit decisions that take place in the analysis activity; indeed, knowledge gained in the design activity may even require revising earlier decisions about the basic feasibility of the project.

Indeed, in the extreme cases, certain events taking place in any activity could cause the entire project to terminate abruptly. The input of management is shown only for the analysis activity, because analysis is the only activity that requires data from management; it is assumed, however, that management exerts *control* over all of the activities.

In summary, then, Figure 5.4 only tells us the input(s) required by each activity and the output(s) produced. The sequence of activities can be implied only to the extent that the presence or absence of data makes it possible for an activity to commence.

5.5 Radical versus conservative top-down implementation

In the previous section, I pointed out that the structured project life cycle allows more than one activity to take place at one time. Let me put it another way: In the most extreme situation, all the activities in the structured project life cycle could be taking place simultaneously. At the other extreme, the project manager could decide to adopt the sequential approach, finishing all of one activity before commencing on the next.

It is convenient to have some terminology to help talk about these extremes, as well as about compromises between the two extremes. A *radical* approach to the structured project life cycle is one in which activities 1 through 9 take place in parallel from the very beginning of the project; that is, coding begins on the first day of the project, and the survey and analysis continue until the last day of the project. By contrast, in a *conservative* approach to the structured project life cycle, all of activity N is completed before activity N + 1 begins.

Obviously, no project manager in his right mind would adopt either of these two extremes. The key to recognize is that the radical and conservative extremes defined above are the two endpoints in a range of choices; this is illustrated in Figure 5.5. There are an infinite number of choices between the radical and conservative extremes. A project manager might decide to finish 75% of the survey activity, followed by completion of 75% of systems analysis, and then 75% of design, in order to produce a reasonably complete skeleton version of a system whose details could then be refined by a second pass through the entire project life cycle. Or, the manager might decide to finish all the survey and analysis activities, followed by completion of 50% of design and 50% of implementation. The possibilities are truly endless!

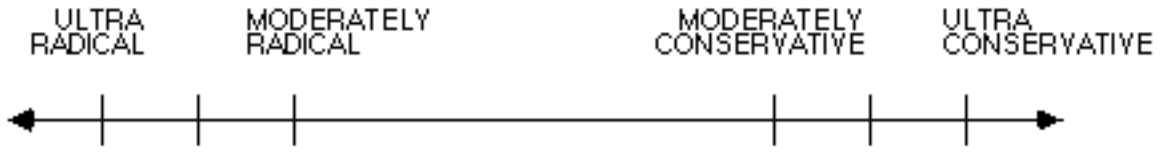

Figure 5.5: Radical and conservative implementation choices

How does a project manager decide whether to adopt a radical or conservative approach? Basically, there is no right answer; the decision is usually based on the following factors:

* How fickle is the user?
* What pressure is the project team under to produce immediate, tangible results?
* What pressure is the project manager under to produce an accurate schedule, budget, and estimate of people and other resources?
* What are the dangers of making a major technical blunder?

As you can appreciate, not one of these questions has a straight black-or-white answer. For example, one can't ask the user of the system, in casual conversation, "By the way, how fickle are you feeling today?" On the other hand, the project manager should be able to assess the situation, based on observation, especially if he or she is a veteran who has dealt with many users and many upper-level managers before.

If the project manager judges that he's dealing with a fickle user — one whose personality is such that he delays final decisions until he sees how the system is going to work — then the manager would probably opt for a more radical approach. The same is true if the manager is dealing with an inexperienced user, who has had very few systems built for him. Why spend years developing an absolutely perfect set of specifications only to discover that the user didn't understand the significance of the specifications?

If, however, the manager is dealing with a veteran user who is absolutely sure of what he wants, and if the user works in a business area that is stable and unlikely to change radically on a month-to-month basis, then the project can afford to take a more conservative approach. Of course, there are a lot of in-between situations: The user may be sure of *some* of the business functions to be performed, but may be somewhat unsure of the kinds of reports and management information he or she would like the system to provide. Or, if the user is familiar with batch computer systems, he or she may be unsure of the impact that an on-line system will have on the business.

Besides fickleness, there is a second factor to consider: the pressure to produce immediate, tangible results. If, due to politics or other external pressures, the project team simply *must* get a system up and running by a specific date, then a somewhat radical approach is warranted. The project manager still runs the risk that the system will be only 90% complete when the deadline arrives, but at least it will be a *working* 90% complete skeleton that can be demonstrated and perhaps even put into production. That's generally better than having finished all the systems analysis, all the design, and all the coding, but none of the testing.

Of course, *all* projects are under some pressure for tangible results; it's simply a question of degree. And it's an issue that can be rather dynamic: A project that begins in a low-key fashion with a comfortable schedule can suddenly become high-priority, and the deadline may be advanced six months or a year. One of the advantages of doing the systems analysis, design, coding, and implementation top-down is that one can stop an activity at any point and leave the remaining details for subsequent consideration; meanwhile, the top-level systems analysis that has been completed can be used to begin the top-level design, and so forth.

Yet another factor in project management is the ever present requirement, in most large organizations, to produce schedules, estimates, budgets, and the like. In some organizations, this tends to be done in a fairly informal fashion, typically because the projects are relatively small, and because management feels that any errors in estimating will have an insignificant impact on the whole organization. In such cases, one can adopt a radical approach, even though any attempts at estimating will have to be "gut-level" guesses. By contrast, most large projects require relatively detailed estimates of personnel requirements, computer resources, and so on; and this can only be done after a fairly detailed survey, analysis, and design have been completed. In other words, the more detailed and accurate the estimates have to be, the more likely the project is to follow a conservative approach.

Finally, the project manager must consider the danger of making a major technical blunder. For example, suppose that all his past experience as a project manager has been with small, one-person Visual Basic projects, which are deployed on a single desktop PC. And now, all of a sudden, he finds himself in charge of developing an Internet-based, multiprocessing distributed database e-commerce system that will process 10 million transactions a day from 50,000 terminals scattered around the world. In such a situation, one of the dangers of a radical approach is discovering a major design flaw after a large portion of the top-level skeleton system has been implemented.

He may discover, for example, that in order for his whizbang system to work, a low-level module has to do its job in 19 microseconds — but his programmers suddenly tell him that there is no way on earth to code the module that efficiently — not in Java, not in C, not even in (ugh!) assembly language. So, he must be alert to the fact that following the radical approach requires his systems analysts and designers to pick a "top" to your system relatively early in the game, and there is always the danger of discovering, down toward the bottom, that they picked the wrong top!

However, consider another scenario: The project manager has decided to build an IT system with new hardware, a new operating system, a new database management system (produced by someone other than the hardware vendor), and a new telecommunications package (produced by yet another vendor). All the vendors have impressive, glossy manuals describing their products, but the vendors have never interfaced their respective hardware and software products together. Who knows if they will work together at all? Who knows if the throughput promised by one vendor will be destroyed by the system resources used by one of the other vendors? Certainly, in a case like this, the project manager might elect a radical approach, so that a skeleton version of the system could be used to explore possible interface and interaction problems between the vendors' components.

If the project manager is in charge of a familiar kind of system, such as her 99th payroll system, then she probably has a very good idea of how realistic her goals are. She probably remembers, from the last project, what sort of modules she's going to need at the detailed level, and she probably remembers very clearly what the top-level system structure looked like. In such a case, she may be willing to accept the risks of making a mistake because of the other benefits that the radical approach will give her.

In summary, the radical approach is most suitable for thinly disguised research and development efforts, where nobody is quite sure what the final system is supposed to do. And it is good in environments in which something *must* be working on a specific date and in situations where the user's perception of what he wants the system to do is subject to change. The conservative approach, on the other hand, tends to be used on larger projects, in which massive amounts of money are being spent and for which careful analysis and design are required to prevent subsequent disasters. However, every project is different and requires its own special blend of radical and conservative top-down implementation. To deal with the individual nature of any project, the project manager must be prepared to modify his approach midstream, if necessary.

5.6 The prototyping life cycle

A variation on the top-down approach discussed above has become popular in quite a few IT organizations. It is generally known as the *prototyping* approach and was introduced in the mid-1980s by Bernard Boar, James Martin, and others. As Boar describes it in (Boar, 1984):

> An alternative approach to requirements definition is to capture an initial set of needs and to implement quickly those needs with the stated intent of iteratively expanding and refining them as mutual user/developer understanding of the system grows. Definition of the system occurs through gradual and evolutionary discovery as opposed to omniscient foresight.

> ... This kind of approach is called prototyping. It is also referred to as system modeling or heuristic development. It offers an attractive and workable alternative to prespecification methods to deal better with uncertainty, ambiguity, and fickleness of real-world projects.

In many ways, this sounds exactly like the radical top-down approach discussed in the section above. The primary difference is that the structured approach discussed throughout this book presumes that sooner or later, a complete *paper model* of the system will be built (i.e., a complete set of dataflow diagrams, entity-relationship diagrams, state-transition diagrams, process specifications, etc.). The model will be completed sooner with a conservative approach and later with a radical approach; but by the end of the project, there will be a formal set of documents that should live forever with the system as it undergoes maintenance and revision.

The prototyping approach, on the other hand, almost always assumes that the model will be a working model, that is, a collection of computer programs that will simulate some or all of the functions that the user wants. But since those computer programs are intended just as a model, there is also an assumption that, when the modeling is finished, *the programs will be thrown away and replaced with REAL programs*. Prototypers typically use the following kinds of software tools:

* An integrated data dictionary
* Screen generator
* Nonprocedural report writer
* High-level, visual programming languages **[12]**
* Nonprocedural query language
* Powerful database management facilities

The prototyping life cycle proposed by Boar is shown in Figure 5.6. It begins with a survey activity, similar to that proposed in this book; this is immediately followed by a determination of whether the project is a good candidate for a prototyping approach. Good candidates for a prototyping approach are projects that have the following characteristics:

* The user is unable (or unwilling) to examine abstract paper models like dataflow diagrams.
* The user is unable or unwilling to articulate (or "prespecify") his or her requirements in any form and can only determine the requirements through a process of trial and error. Or, as my colleague Bob Spurgeon puts it, this is the situation where the user says, "I don't know what I want, but I'll recognize it when I see it!"
* The system is intended to be on-line with full-screen terminal activities, as opposed to batch edit, update, and report systems. (Almost all prototyping software tools are oriented toward the on-line, database-driven, terminal-oriented approach; there are few vendor-supplied software tools to help build prototypes of batch systems.)
* The system does *not* require specification of large amounts of algorithmic detail, that is, the writing of many, many process specifications to describe the algorithms by which output results are created. Thus, decision support, ad hoc retrieval, and record management systems are good candidates for prototyping. Good candidates tend to be the systems in which the user is more concerned about the format and layout of the CRT data entry and output screens and error messages than about the underlying computations performed by the system.

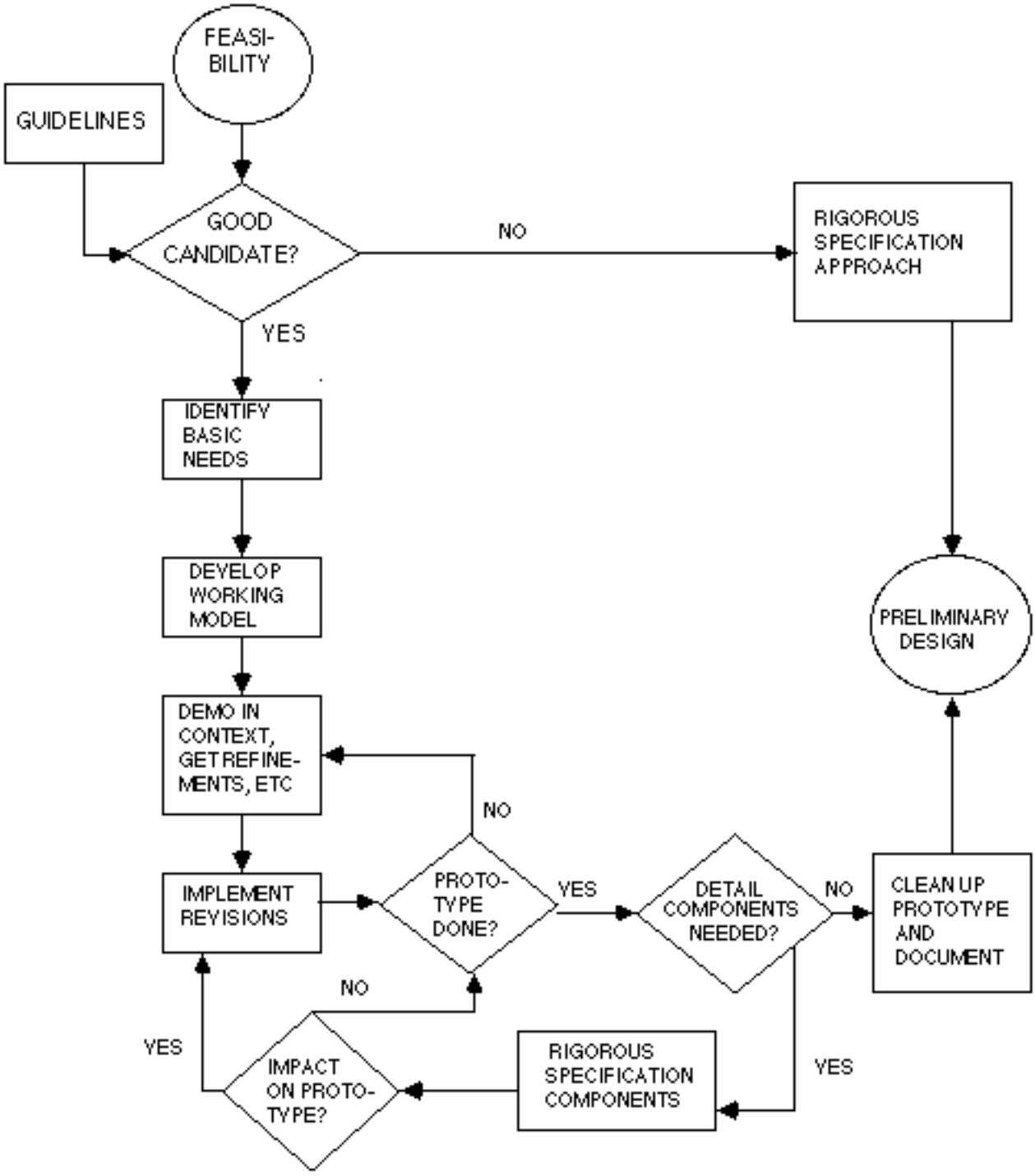

Figure 5.6: The prototyping life cycle

It is significant to note that the prototyping life cycle shown in Figure 5.6 concludes by entering the design phase of a "traditional" structured life cycle of the sort described in this book. Specifically, this means that the prototype is *not* intended to be an operational system; it is intended solely as a means of modeling user requirements.

The prototyping approach certainly has merit in a number of situations. In some cases, the project manager may want to use the prototyping approach as an alternative to the structured analysis approach described in this book; in other cases, she or he may wish to use it *in conjunction with* the development of paper models such as dataflow diagrams. Keep in mind the following points:

* The top-down approach described in the previous section is another form of prototyping, but instead of using vendor-supplied tools, such as screen generators and fourth generation languages, the project team uses the system itself as its own prototype. That is, the various versions of a skeleton system provide the working model that the user can interact with to get a more realistic feeling of system functions than he or she might get from a paper model.
* The prototyping life cycle, as described above, involves the development of a working model that is then thrown away and replaced by a production system. There is a significant danger that either the user or the development team may try to turn the prototype itself into a production system. This usually turns out to be a disaster because the prototype cannot handle large volumes of transactions efficiently, and because it lacks such operational details as error recovery, audit trails, backup/restart facilities, user documentation, and conversion procedures.
* If the prototype is indeed thrown away and replaced by the production system, there is a real danger that the project may finish without having a permanent record of user requirements. This is likely to make maintenance increasingly difficult as time goes on (i.e., ten years after the system is built, it will be difficult for maintenance programmers to incorporate a change, because nobody, including the "second-generation" users who are now working with the system, will remember what it was supposed to do in the first place). The life cycle presented in this book is based on the idea that the paper models developed in the analysis activity will not only be input to the design activity, but will also be retained (and modified, as necessary) during ongoing maintenance. In fact, the models may survive the system that implements them and serve as a specification for the replacement system.

5.7 Summary

The major purpose of this chapter has been to provide an overview of project life cycles in general. If you examine the formal project life cycle in any systems development organization, you should be able to tell whether it falls into the category of classical, semistructured, structured, or prototyping.

If your projects only allow one activity at a time, the discussion of radical top-down implementation and conservative top-down implementation in Section 5.6 may have disturbed you. This was my intent, as the major purpose of that section was to make you think about the *possibility* of overlapping some of the major activities in a systems development project. Obviously, it's more difficult to manage any project that has several activities taking place in parallel — but to some extent, that always happens in every project. Even if the project manager decides that his or her people will concentrate all their efforts on one major activity at a time, there will still be a number of subactivities taking place in parallel. Multiple systems analysts will be interviewing multiple users simultaneously; various pieces of the final product of systems analysis will be in various stages of progress throughout the analysis phase. One job of the project manager is to exercise sufficient control over those subactivities to ensure that they coordinate smoothly. And in virtually every IT project, this same kind of parallel activity operated at a higher level, too; that is, despite what the organization's formal project life cycle may have recommended, the reality is that many of the major project activities do overlap to some extent. Nevertheless, if the project manager decides to insist on a strictly sequential progression of project activities, the project life cycle presented in this book will still work.

References

1. James A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House, 2000.

2. Kent Beck, *eXtreme Programming eXplained*. Addison Wesley, 2000.

3. Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals and Applications in Software Engineering*, 2nd ed. Englewood Cliffs, NJ: YOURDON Press, 1989.

4. Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed. Englewood Cliffs, NJ: YOURDON Press, 1988.

5. Bernard Boar, *Application Prototyping*. Reading, MA.: Addison-Wesley, 1984.

6. James Grier Miller, *Living Systems*. New York: McGraw-Hill, 1978.

7. Edward Yourdon, *Managing the Systems Life Cycle*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

8. Winston W. Royce, "Managing the Development of Large Software Systems." *Proceedings, IEEE Wescon*, August 1970; pp. 1-9.

9. Barry Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

10. Bill Inmon, *Information Engineering for the Practitioner: Putting Theory into Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

11. Marvin Gore and John Stubbe, *Elements of Systems Analysis*, 3rd ed. Dubuque, IA: William Brown, 1983.

Endnotes

1. This sounds as if anarchy prevails in most IT organizations; however, there are two common situations that lead to this individualistic approach even in the most exemplary organization: (1) the highly decentralized organization, where every department has its own IT group with its own local standards, and (2) the period of several years immediately after the last "official project life cycle" was deemed a failure and thrown out.

2. There are several such packages on the market, costing from $10,000 to $100,000 or more; the larger "Big-5" accounting firms, and major IT consulting organizations have developed proprietary versions of such packages, which they sometimes sell together with consulting services and automated tool-support. I won't comment on any specific project management package; I will only suggest that you keep the concepts presented in this book in mind if your organizations uses a vendor-supplied package.

3. Miller points out in (Miller, 1978) that this is a commonly observed phenomenon; indeed, he presents it as a general "hypothesis" applicable to all living systems: "HYPOTHESIS 2-1: System components incapable of associating, or lacking experience which has formed such associations, must function according to rigid programming or highly standardized operating rules. It follows that as turnover of components rises above the rate at which the components can develop the associations necessary for operation, rigidity of programming increases."

4. In fact, the politics of most IT development projects are such that there is only one checkpoint at which the user has an obvious, clean way of backing out: at the end of the survey, or feasibility study, phase. In theory, though, the user should have the opportunity to cancel an IT project at the end of any phase if he thinks he is wasting money.

5. Many people feel that the bottom-up approach may also have come from the computer hardware industry, because many of the early computer programmers and programming managers in the 1950s and 1960s were electrical engineers who had previously been involved in the development of hardware.

6. I'm convinced that yet another of the Murphy-type laws applies in this regard: The larger and more critical the project, the more likely it is that its deadline will coincide with end-of-year processing and other organizational crises that gobble up all available computer time!

7. We will summarize these modern development techniques in Chapter 7.

8. Interviewing techniques are discussed in Appendix E.

9. The context diagram is part of the environmental model that we will discuss in detail in Chapter 18. Its major purpose, as indicated here, is to define the scope of the system (what is in the system and what is out of the system), as well as the various terminators (people, organizational units, other computer systems, etc.) with whom the system will interact.

10. Cost-benefit calculations are discussed in Appendix C.

11. Actually, there are ways of showing feedback and control in dataflow diagrams, as we will see in Chapter 9. The additional notations (for control processes and control flows) are normally used for modeling real-time systems, and we have avoided their use in this model of the "system for building systems."

12. In Chapter 23, we refer to these as "5th generation" languages, to distinguish them from the older 4th generation languages which provided very high-level capabilities, but still required programmers to use "line-at-a-time" text editors to create the programs. Examples of visual development environments are Microsoft's Visual Basic and Visual C++; IBM's Visual Age for Smalltalk and its visual COBOL development environment; and most vendor-supplied versions of Java.

Questions and Exercises

1. List two synonyms for methodology.

2. What is the difference between a tool, as used in this book, and a methodology?

3. What are the three major purposes of a project life cycle?

4. *Research Project*: Find the price of three commercial methodology products offered by software vendors or consulting firms.

5. Why do smaller data processing organizations typically not use formal methodologies?

6. Why is a methodology useful for new managers?

7. Why is it important to have a methodology in an organization with many different projects underway?

8. Why is a methodology useful for controlling projects?

9. What are the major distinguishing characteristics of the classical life cycle?

10. What does bottom-up implementation mean?

11. What are the four major difficulties with the bottom-up implementation strategy?

12. What kind of environment is suitable for a bottom-up implementation approach?

13. Why does it matter that "nothing is done until it's all done," which characterizes the bottom-up approach?

14. Why should trivial bugs be found first in the testing phase of a project?

15. What is the difference between testing and debugging?

16. Why is debugging difficult in a bottom-up implementation?

17. What is meant by the phrase sequential progression when describing a project life cycle?

18. What are the two major problems with sequential progression?

19. What are the main differences between the classical and the semistructured life cycle?

20. What are the two major consequences of the top-down implementation approach?

21. Why does the design activity in the semistructured life cycle often involve redundant work?

22. What are the major differences between the structured and semistructured life cycle?

23. List the nine activities of the structured project life cycle.

24. Who are the three types of people who provide primary input to the project life cycle?

25. What are the five major objectives of the survey activity?

26. What is a context diagram?

27. What is the major purpose of the analysis activity?

28. What are the types of models produced by the analysis activity?

29. What is the purpose of the design activity?

Questions and Exercises (cont.)

30. What are the two major issues that the user is typically concerned with in the design activity?

31. When can acceptance test generation (Activity 5) begin?

32. What is the purpose of the procedure description activity?

33. Why has a DFD been used in Figure 5.4 to show the structured project life cycle?

34. What is a possible synonym for activity?

35. Why is feedback important in the structured project life cycle?

36. What is the difference between the radical and conservative approach to the structured project life cycle?

37. What are the four main criteria for choosing the radical approach versus the conservative approach?

38. Can you think of any additional criteria that might be used for choosing a radical approach versus a conservative approach?

39. What kind of approach (radical versus conservative) should a project manager choose if the user is likely to change his or her mind about the requirements of the system?

40. What kind of approach (radical versus conservative) should a project manager choose if the project is under severe time pressure?

41. What kind of approach (radical versus conservative) should a project manager choose if the project faces major technical risks?

42. What is the difference between the prototyping life cycle and the radical life cycle?

43. What characteristics does the ideal prototyping project have?

44. What kind of tools are typically needed for a prototyping project?

45. Why are batch systems generally not good candidates for prototyping projects?

46. What are the dangers of the prototyping approach?

47. How can the prototyping approach and the structured project life cycle be used together in a project?