Unfortunately, Babbage never completed construction of either of his machines. Some historians believe that he never finished them because the technology of the period could not support such complex machinery. But most feel that Babbage's failure was his own doing. He was both brilliant and somewhat eccentric (it is known that he was afraid of Italian organ grinders, for example). As a consequence, he tended to abandon projects midway through so that he could concentrate on newer and better ideas. He always believed that his new approaches would let him complete a machine in less time than if he continued with his old ideas.

Thus when he died, Babbage had numerous pieces of computing machines and partial drawings of designs, but none of the plans was sufficiently complete to produce a single working computer. After his death, his inventions were mostly dismissed and ignored until modern computers were developed. Only then did historians recognize the true importance of his contributions. Babbage stumbled upon the idea of the computer a full century before it was developed. Today we can only imagine how different the world would be had he succeeded.

Programming Methodology

The programming process consists of a problem-solving phase and an implementation phase. In Chapter 1 we discussed some strategies for solving problems, and in Chapter 2 we saw how some simple programs are implemented. Here we describe a methodology for developing data models and algorithmic solutions for more complex problems. This methodology will help you write algorithms that are easy to implement as Ada programs and, consequently, programs that are readable, understandable, and easy to debug and modify.

Top-Down Design

The technique we use is known as **top-down design** (it's also called *stepwise refinement* and *modular programming*). It allows us to use the divide-and-conquer approach that we talked about in Chapter 1.

Top-down design A technique for developing a program in which the problem is divided into more easily handled subproblems, the solutions of which create a solution to the overall problem.

In top-down design, we work from the abstract (a list of the major parts of a solution) to the particular (data types and algorithmic steps that can be translated directly into Ada code). You also can think of this as working

from a high-level solution that leaves the details of implementation unspecified down to a fully detailed solution.

The easiest way to solve a problem is to give it to someone else and say, "Solve this problem." This is the most abstract level of a problem solution—a single-statement solution that encompasses the entire problem without specifying any of the details of implementation. It's at this point that programmers are called in. Our job is to turn this abstract solution into a concrete solution—a program.

We start by breaking the solution into a series of major steps. In the process, we move to a lower level of abstraction—some of the implementation details are now specified. Each of the major steps becomes an independent subproblem that we can work on separately. In a very large project, one person (the *chief architect* or *team leader*) would formulate the subproblems and then give them to other members of the programming team, saying "Solve this problem." In the case of a small project, we just give the subproblems to ourselves. Then we choose one subproblem at a time and break it into another series of smaller subproblems. The process continues until each subproblem can be solved directly.

Why do we work this way? Why not simply write out all of the details? Because it is much easier to focus on one problem at a time. For example, suppose you are working on a program to print out certain values and discover that you need a complex formula to calculate an appropriate width parameter for one of them. Calculating widths is not the purpose of the program. If you shift your focus to the calculation, you are more likely to forget some detail of the printing process. What you do is write down an abstract step—"Calculate the width required"—and go on with the problem at hand. Once you've completed the general solution, you can go back to solving the step that does the calculation.

By subdividing the problem, you create a hierarchical structure called a *tree structure*. Each level of the tree is a complete solution to the problem that is less abstract than the level above it. Figure 3-4 shows a solution tree for a problem. Steps that are shaded have enough implementation details specified to be translated directly into Ada statements. These are **concrete steps.** Those that are not shaded are **abstract steps;** they reappear as subproblems in the next level down. Each box represents a **module.** Modules are the basic building blocks of top-down solutions. The diagram in Figure 3-4 is also called a *module structure chart.*

Concrete step A step for which the implementation details are fully specified.

Abstract step A step in which some implementation details remain unspecified.

Module A self-contained collection of steps that solves a problem or subproblem; can contain both concrete and abstract steps.

This module is often named "main"

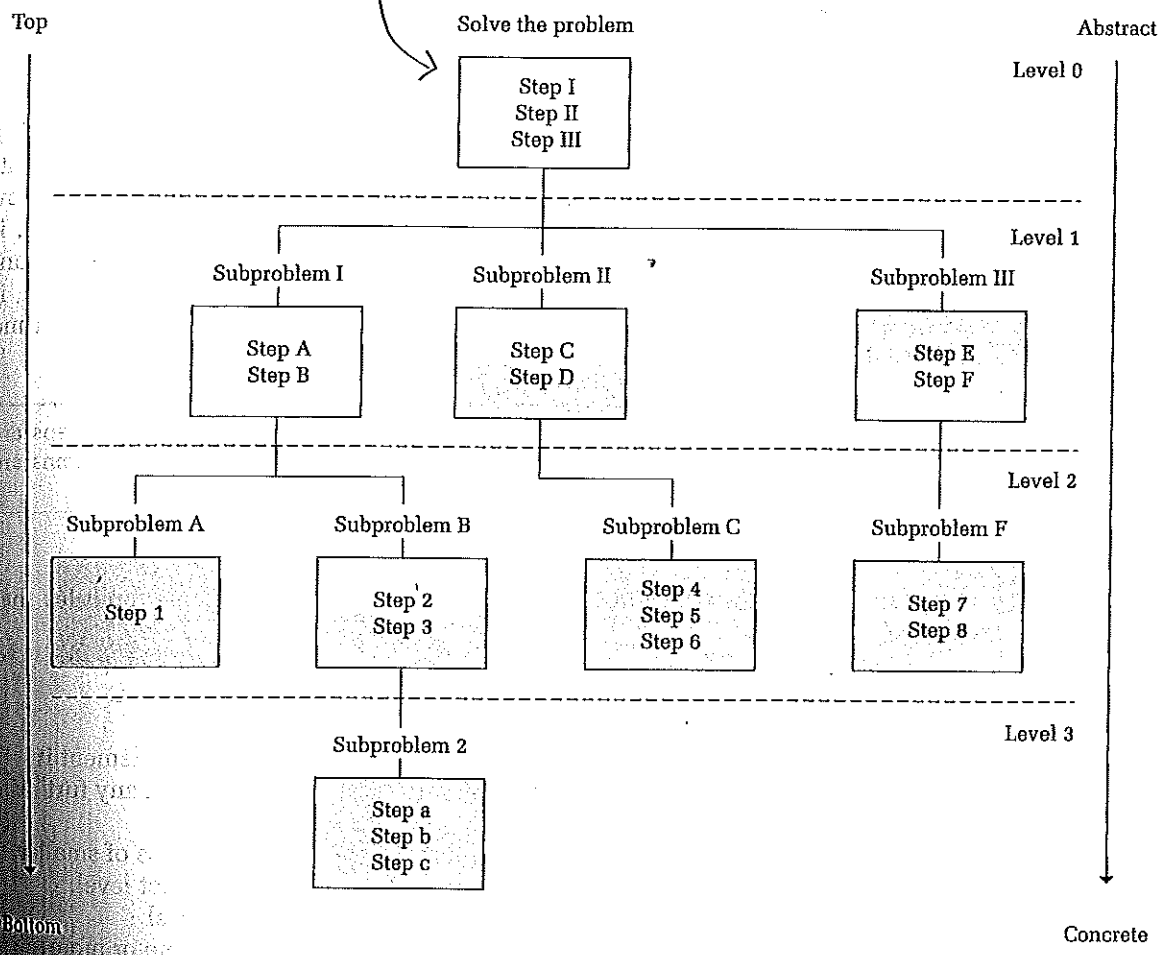

FIGURE 3-4
Hierarchical Solution Tree

Modules A module begins life as an abstract step in the next higher level of the solution tree. It is completed when it solves a given subproblem: when it specifies a series of steps that does the same thing as the higher-level abstract step. At this stage a module is **functionally equivalent** to the abstract step.

Functional equivalence A property of a module—it performs exactly the same operation as the abstract step it defines. A pair of modules are functionally equivalent to each other if they each accomplish the same abstract operation.

A properly designed module contains only concrete steps that directly address the given subproblem and abstract steps for significant new subproblems. This is called **functional cohesion.** The idea behind functional cohesion is that each module should do just one thing and do it well. Functional cohesion is not a well-defined property; there is no quantitative measure of cohesion. It is a product of the human need to organize things into neat chunks that are easy to understand and remember. Knowing which details to make concrete and which details to leave abstract is a matter of experience, circumstance,,and personal style. For example, you might decide to include a field width calculation in a printing module, if there isn't too much detail in the rest of the module so that it becomes confusing. On the other hand, if the calculation is performed several times, it makes sense to write it as a separate module and just refer to it each time you need it.

Note: Each module in the tree can map to a function in our Python code!

Functional cohesion A property of a module in which all concrete steps are directed toward solving just one problem, and any significant subproblems are written as abstract steps.

Writing cohesive modules Here's one approach to writing modules that are cohesive.

1. Think about how you would solve the subproblem by hand.
2. Begin writing down the major steps.
3. If a step is simple enough so that you can see how to implement it directly in Ada, it is at the concrete level; it doesn't need any further refinement.
4. If you have to think about implementing a step as a series of smaller steps or as several Ada statements, it is still at an abstract level.
5. If you are trying to write a series of steps and start to feel overwhelmed by details, you are probably bypassing one or more levels of abstraction. Stand back and look for pieces that you can write as more abstract steps.

We could call this the "procrastinator's technique." If a step is cumbersome or difficult, put it off to a lower level; don't think about it today, think about it tomorrow. Of course tomorrow does come, but the whole process then can be applied to the subproblem. A trouble spot often seems much simpler when you can focus on it. And eventually the whole problem is broken down into manageable units.

As you work your way down the solution tree, you make a series of design decisions. If a decision proves awkward or wrong (and many times it will!), it's easy to backtrack (go back up the tree to a higher-level mod-

ule) and try something else. You don't have to scrap your whole design—only the small part you are working on. There may be many intermediate steps and trial solutions before you reach a final design.

The modules developed for the case studies throughout this book are presented as though we wrote them down that way the first time. Nothing could be further from the truth! The designs shown are the final product of a long process of trying and discarding many different ones. To show all of the intermediate attempts we made would easily double the size of this text. So don't hesitate to throw out a design and begin again. And don't be discouraged if it takes you a number of attempts to achieve a design. The problem-solving phase of the programming process takes time. If you spend the bulk of your time analyzing and designing a solution, coding and implementing the program will take very little time.

You'll find it easier to implement a design if you write the steps in pseudocode. *Pseudocode* is a mixture of English statements and Ada-like control structures that can easily be translated into Ada. (We've been using pseudocode in the algorithms in the Problem-Solving Case Studies.) When a concrete step is written in pseudocode, it should be possible to rewrite it directly as a statement in a program.

Implementing a Design The product of top-down design is a hierarchical solution to a problem with multiple levels of abstraction. Figure 3-5 shows the top-down design we developed for program Quilt. This kind of solution forms the basis for the implementation phase of programming.

How do we translate a top-down design into an Ada program? If you look closely at Figure 3-5, you can see that we can assemble the concrete steps (those that are shaded) into a complete algorithm for solving the problem. Their position in the tree determines the order in which they are assembled. We start at the top of the tree, at level 0, with the first step "Compute Fill Volume." Because it is abstract, we must go to the next level, level 1. There we find a series of steps that correspond (are functionally equivalent) to this step. The first of these steps, "Compute Tube Volume," is abstract, so we must go to the next level, level 2. There we find a corresponding series of concrete steps; this series of steps becomes the first part of our algorithm. Because the conversion process is now concrete, we can go back a level to level 1 and go on to the next step, calculating the Fill Volume. This step is concrete; we can copy it directly into the algorithm. Now we are ready to return to level 0. The last three steps at level 0 are abstract, so we work with each of them in order at level 1, making them concrete. Here's the resulting nonhierarchical algorithm.