CHAPTER SEVEN

# PROGRAMMING IN THE LARGE

*"Human fallibility—from grand to grandiose"*

*(Mills 1979)*

The production of large programs—those consisting of more than several thousand lines—presents many challenging problems that do not arise when developing smaller programs. The same methods and techniques that work well with small programs do not necessarily apply to larger programs. As the lack of applicability of this "scaling up" has been recognized, approaches to the production of large programs have been pursued along three paths.

1. Management techniques to provide control over personnel assigned to a software project and monitor their progress.
2. Software design methodologies to attack the particular problems found in large programs. These methodologies have in turn led to the development of supporting language facilities.
3. Software development tools to assist in the creative part and to automate as much as possible the noncreative parts of software development.

To stress the differences between small and large systems production, DeRemer and Kron invented the terms "programming in the small" and "programming in the large." Point 1 above reflects the most striking difference between programming in the small and programming in the large. If only one person can produce the needed software, all problems boil down to the professional skill of that one programmer and the availability of a suitable programming environment. The development of large software systems which requires several programmers, on the other hand, is not merely a matter of "programming," but mainly a management problem. Successful management must be capable of estimating the resources necessary to accomplish a given

task, assigning resources to the project at the appropriate times, breaking down the development effort into well-defined and clearly separated phases, monitoring each phase through periodic design reviews, and imposing a set of standards for each activity. Aron notes that the "emphasis on management rather than technology represents a major change in the nature of programming since the 1950s" (Aron 1974). Most of these issues, however, are beyond the scope of this book; the interested reader is referred to the literature referenced in the Further Reading section.

This chapter focuses on points 2 and 3 given above. The two points are strongly related. For example, a program library can be viewed either as simply a tool or as a necessary component of a methodology based on the incremental development and production of reusable software.

The chapter is organized as follows. Section 7.1 further motivates the distinction between programming in the small and programming in the large, with the aid of an example. Section 7.2 reviews software design methodologies and stresses the differences between design methodologies for small and large programs. Section 7.3 evaluates the features provided by current languages in support of programming in the large. Section 7.4 discusses the need for program development systems in which the language and a rich set of tools are integrated to provide a friendly programming environment.

## 7.1 WHAT IS A LARGE PROGRAM?

The concept of a large program is difficult to define. We certainly do not want to equate the size of a program (e.g., the number of source statements) with its complexity. Largeness relates more to the "size" and complexity of the problem being solved than to the final size of a program. Usually, however, the size of a program is a good indication of the complexity of the problem being solved.

Consider the task of building a reservation system for a particular airline. The system is expected to keep a data base of flight information. Airline agents working at remote sites may access the data base at arbitrary times and in any order. They may inquire about flight information, such as time and price; make or cancel a reservation on a particular flight; update existing information, such as local telephone number for a passenger. Certain authorized personnel can access the data base to do special operations, such as adding or canceling a flight, or changing a plane type. Others may access the system to obtain statistical data about a particular flight or all flights.

A problem of this magnitude imposes severe restrictions on the solution strategy to be followed. The characteristics of such problems include the following.

- The system has to function correctly. A seemingly small error, such as losing a reservation list or interchanging two different lists, could be extremely costly. To guarantee correctness of the system virtually any cost can be tolerated.



- The system is long-lived. The cost associated with producing such a system is so high that it is not practical to replace it with a totally new system. It is expected that the cost will be recouped only over a long period of time.
- During its lifetime, the system undergoes considerable modification. For our example, because of completely unforeseen new Federal regulations, changes might be required in price structure, a new type of airplane might be added, and so on. Other changes might be desirable because experience with the system has uncovered some weaknesses. We might find it desirable to have the system find the best route automatically by trying different connections.
- Because of the magnitude of the problem, many people—tens or hundreds—are involved in the development of the system.

These characteristics impose severe requirements both on the system development process and on the tools used, namely

- The work must be divided among different people. The work assigned to each person must be stated clearly and unambiguously. One person should not have to know the details of other persons' work, just how theirs interacts with his or hers.
- The system is built up from pieces developed independently by several people. We will call such pieces *modules*. These modules must be designed and certified independently. It would be beneficial if some of these pieces could be taken from already-existing systems. Similarly, it would be beneficial if these pieces could be reused in future projects. Indeed, the application itself is so general that the entire system might be reused, with minor modifications, for different airlines.
- The system must be modifiable easily: that is, it should be possible to change the internals of one module without requiring changes to the entire system.
- It must be possible to show the correctness of the system based on the correctness of the constituent modules.

These characteristics illustrate the concept of a large program. These problems are less important with a one-person programming task: For example, there would be less need for dividing the work and thus less emphasis on clear interface specifications. The possibility exists, at least in theory, of redoing the system from scratch, and so modifiability is less important. Correctness is involved only with one module and therefore is easier to assess. On the other hand, a large system is composed of a collection of interacting modules. If no systematic design methodologies are adopted, each module can interact with any other module in some subtle manner. Consequently,

each module cannot be designed, understood, and proven correct apart from all the other modules. The complexity of such systems becomes unmanageable.

The boundaries between programming in the large and programming in the small can hardly be stated rigorously. However, we can assume that programming in the large addresses the problem of modular system decomposition, whereas programming in the small refers to the production of individual modules. The methodologies useful for the two activities are discussed in Section 7.2. Section 7.3 evaluates programming languages in light of programming in the large.

## 7.2 PROGRAMMING IN THE SMALL VERSUS PROGRAMMING IN THE LARGE: DESIGN METHODOLOGIES

Research in software design methodologies has focused on top-down design as an effective way of mastering the difficulties of software production. In top-down design, a problem is iteratively decomposed into subproblems that can be solved independently.

When applied to small programs, the method is named *stepwise refinement*. Stepwise refinement has been illustrated in the literature as a process of writing and rewriting a program. At each step of development, the program consists of declarations and statements, some of them legal in the programming language, others abstract and to be refined at the next step. At each step of refinement, abstract declarations and statements can be left in the code as comments for the purpose of documentation. The process continues until eventually the entire text is a legal program.

Although stepwise refinement is an effective methodology for programming in the small, it fails when applied to large programs. One reason is that it does not favor the recognition of commonalities between parts. Programmers are not encouraged to represent common parts by a unique abstraction, to be refined just once and invoked wherever necessary. Rather, each part is separately refined. Another reason is that the final program does not mirror the design process adequately, even if abstract statements are left as comments in the text. Moreover, abstract statements usually are written in informal English prose, and it may be necessary to read their refinement to understand precisely what they do.* Consequently, readability and modifiability of programs can be hampered for programs of substantial size.

Top-down design of large applications must support system decomposition into small-sized programs (modules). Following the principle of information hiding, the designer must distinguish clearly between what a module does—what the module exports for use by other modules—from how it does it—its internal details. A *module interface* should specify exactly the internally defined entities exported for use by other modules and the externally defined entities imported from other modules. Designing a module consists of designing its interface and, iteratively, any new subsidiary modules that will be used by the module.

*This last problem can be solved by stating the abstract statements in a formal notation.



As an example, consider the airline reservation problem of Section 7.1. We might have a *flight module*, which provides information about flights as they are scheduled. Given a flight number, the module provides the total number of seats, the time of departure, possible connecting flights, etc. The module also provides update operations—for example, to modify departure times. We might have a *reservation module* to handle reservation lists for all flights; given a flight number and a date, it allows one to make or cancel reservations on a particular flight. A *statistics module* might provide operations that collect statistics on some or all flights. The reservation module must have a restricted access to the flight module. It can obtain information about flights (e.g., number of seats) but cannot perform any updates. For example, the design of the reservation module can be recorded as sketched below.

**module** *reservation*
    **import function** *number_of_seats (flight number)* **return** *integer*
        **from** *flight;*
    **export procedure** *make_reservation (flight_number, customer);*
        **procedure***cancel_reservation (flight_number, customer);*

Function *number_of_seats* is imported from module *flight;* it receives a formal parameter of type *flight_number* and returns an *integer*. Procedures *make_reservation* and *cancel_reservation* have parameters of type *flight_number* and *customer;* they update the list of passengers for a certain flight.

Design is complete when all module interfaces have been specified. Only at this stage can we turn to the implementation of module bodies. The separation between the two phases of design and implementation distinguishes this approach from stepwise refinement. In stepwise refinement, design and coding are developed hand in hand. Here we first decompose the system into modules, then later address the problem of implementing the module bodies (e.g., using stepwise refinements).

It can be argued that modular system decomposition can be driven by the recognition of two particular classes of abstractions: procedural abstractions and data abstractions. Procedural abstractions are operations that perform a mapping between input and output data objects. Data abstractions are a set of operations that manipulate a particular class of data objects. A module corresponds either to a procedural abstraction or to a data abstraction. At each stage of top-down design, the problem to be solved is how to design an abstraction. Each abstraction is iteratively decomposed into subsidiary abstractions until the program is broken into pieces of moderate complexity.

## 7.3 LANGUAGE FEATURES FOR PROGRAMMING IN THE LARGE

This section discusses how programming languages support the needs of programming in the large and, in particular, the design methodology presented in Section 7.2. All programming languages provide features for decomposing programs into smaller and

largely autonomous units. Such units will be called *physical modules* in the sequel; we will use the term *logical module* to denote a module identified at the design stage. A logical module may be implemented by one or more physical modules.

Our discussion will be organized according to the following three criteria.

1. What is the notion of physical module supported by the language, and how well does it capture the logical properties specified at the design stage?
2. How can a program be built by combining physical modules? How does the program structure imposed by the language mirror the hierarchical modular decomposition found during design?
3. How independently can physical modules be implemented? In particular, how long-lived and reusable are physical modules?

The discussion will be centered around Pascal, SIMULA 67, and Ada. Pascal can be viewed here as a representative of the class of ALGOL-like languages. Our conclusions about Pascal hold, with minor changes, for other members of the class, such as ALGOL 60 and ALGOL 68. A few comments on CLU will be given in the section devoted to Ada.