A] STRUCTURING THE PROGRAM:

1. We describe the basic concepts for structuring large programs (encapsulation, interfaces, information hiding) and the mechanisms provided by languages to support it (packaging, separate compilation).
2. We also consider the concept of genericity in building software component libraries. We do not go deeply into object-oriented programming, which is the subject of the next chapter. The production of large programs—those consisting of more than several thousand lines—presents challenging problems that do not arise when developing smaller programs.
3. The same methods and techniques that work well with small programs just don't "scale up." To stress the differences between small and large systems production, we refer to "programming in the small" and "programming in the large."
4. Two fundamental principles abstraction and modularity—underlie all approaches to programming in the large. Abstraction allows us to understand and analyze the problem by concentrating on its important aspects. Modularity allows us to design and build the program from smaller pieces called modules.
5. During problem analysis, we discover and invent abstractions that allowing languages organize data through the concept of type. Types are used as a way to classify data according to different categories. us to understand the problem. During program design and implementation, we try to discover a modular structure for the program.
6. In general, if modules that implement the program correspond closely to abstractions discovered during problem analysis, the program will be easier to understand and manage. The principles of modularity and abstraction help us apply the wellknown problem solving strategy known as "divide and conquer."
7. The concept of a "large program" is difficult to define precisely. 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 in terms of the number of source lines. Often, however, the size of a program is a good indication of the complexity of the problem being solved.

8. Consider the task of building an airline reservation system. The system is expected to keep a database of flight information. Reservation agents working at remote sites may access the database 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 the local telephone number for a passenger.
9. Certain authorized personnel can access the database to do special operations, such as adding or canceling a flight, or changing the type of the airplane assigned to a flight. 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 and the following key requirements:

- The system has to function correctly. A seemingly small error, such as assignment to the wrong pointer, may lead to losing a reservation list or interchanging two different lists and 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 government regulations, changes might be required in price structure, a new type of airplane might be added, and so on. Other changes might be considered because experience with the system has uncovered new requirements. 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.

B] Software design methods:

1. To combat the complexities of programming in the large, we need a system atic design method that guides us in composing a large program out of smaller units—which we call modules.

2. A good design is composed of modules that interact with one another in well-defined and controlled ways. Consequently, each module can be designed, understood, and validated independently of the other modules.
3. Once we have achieved such a design, we need programming language facilities that help us in implementing these independent modules, their relationships, and their interactions.
4. The goal of software design is to find an appropriate modular decomposition of the desired system. Indeed, even though the boundaries between programming in the large and programming in the small cannot be stated rigorously, we may say that programming in the large addresses the problem of modular system decomposition, and programming in the small refers to the production of individual modules.
5. A good modular decomposition is one that is based on modules that are as independent from each other as possible. There are many methods for achieving such modularity. A well-known approach is information hiding which uses the distribution of "secrets" as the basis for modular decomposition.
6. Each module hides a particular design decision as its secret. The idea is that if design decisions have to be changed, only the module that "knows" the secret design decision needs to be modified and the other modules remain unaffected.
7. If a design is composed of highly independent modules, it supports the requirements of large programs:Independent modules form the basis of work assignment to individual team members. The more independent the modules are, the more independently the team members can proceed in their work.

- The correctness of the entire system may be based on the correctness of the individual modules. The more independent the modules are, the more easily the correctness of the individual modules may be established.
- Defects in the system may be repaired and, in general, the system may be enhanced more easily because modifications may be isolated to individual modules.

C] Concepts in support of modularity:

1. To summarize the discussion of the last section, the key to software design is modularization. A good module represents a useful abstraction; it interacts with other modules in well-defined and regular ways; it may be understood, designed, implemented,

compiled, and enhanced with access to only the specification (not the implementation secrets) of other modules.

2. Programming languages provide facilities for building programs in terms of constituent modules. In this chapter, we are interested in programming language concepts and facilities that help the programmer in dividing a program into subparts—modules—the relationships among those modules and the extent to which program decompositions can mirror the decomposition of the design.Procedures and functions are an effective way of breaking a program into two modules: one which provides a service and another which uses the service.
3. We may say that the procedure is a server or service provider and the caller is a client. Even at this level we can see some of the differences between different types of modularization units. For example, if we provide a service as a function, then the client has to use the service in an expression.
4. On the other hand, if we provide the service in a procedure, then the client may not use it in an expression and is forced to use a more assignment-oriented or imperative style. Procedures and functions are units for structuring small programs, perhaps limited to a single file. Sometimes, we may want to organize a set of related functions and procedures together as a unit.
5. Ada and Modula-2 provide other constructs for this purpose. Before we delve into specific language facilities, we will first look at some of the underlying concepts of modularity. These concepts help motivate the need for the language facilities and help us compare the different language approaches.

D] Encapsulation:

1. A program unit provides a service that may be used by other parts of the program, called the clients of the service. The unit is said to encapsulate the service. The purpose of encapsulation is to group together the program components that combine to provide a service and to make only the relevant aspects visible to clients.
2. Information hiding is a design method that emphasizes the importance of concealing information as the basis for modularization. Encapsulation mechanisms are linguistic constructs that support the implementation of information hiding modules. Through

encapsulation, a module is clearly described by two parts: the specification and the implementation.

3. The specification describes how the services provided by the module can be accessed by clients. The implementation describes the module's internal secrets that provide the specified services.
4. For example, assume that a program unit implements a dictionary data structure that other units may use to store and retrieve <name, "id"> pairs. This dictionary unit makes available to its clients operations for: inserting a pair, such as <"Mehdi", 46>, retrieving elements by supplying the string component of a pair, and deleting elements by supplying the string component of a pair.
5. The unit uses other helper routines and data structures to implement its service. The purpose of encapsulation is to ensure that the internal structure of the dictionary unit is hidden from the clients.
6. By making visible to clients only those parts of the dictionary unit that they need to know, we achieve two important properties.

• The client is simplified: clients do not need to know how the unit works in order to be able to use it; and

• The service implementation is independent of clients and may be modified without affecting the clients.

7. Different languages provide different encapsulation facilities. For example, in C, a file is the unit of encapsulation. Typically, the entities declared at the head of a file are visible to the functions in that file and are also made available to functions in other files if those functions choose to declare them. The declaration:

extern int max;

8. states that the variable max to be used here, is defined—and storage for it allo-cated—elsewhere.

E] Interface and implementation:

1. A module encapsulates a set of entities and provides access to some of those entities. The available entities are said to be exported by the module. Each of the exported entities is available through an interface.

2. The collection of the interfaces of the exported entities form the module interface. Clients request the services provided by a module using the module's interface, which describes the module's specification.
3. The interface specifies the syntax of service requests. Some languages also support or require the specification of the interface's semantic requirements. The idea is that the interface is all that the client needs to know about the provider's unit. The implementation of the unit is hidden from the client.
4. The separation of the interface from the implementation contributes to the independence of the client and the server from one another. A service provider exports a set of entities to its clients. A client module imports those entities to be able to use the services of the provider module.
5. The exported entities comprise the service provided by the module. Some languages have implicit and others explicit mechanisms for import and export of entities. Languages also differ with respect to the kinds of entities they allow to be exported.
6. For example, some languages allow a type to be exported and others do not. A function declaration such as:

```
int max (int& x, int& y)
```

7. specifies to the clients that the function max may be called by passing to it two integers; the function will return an integer result. We introduced the term signature to refer to these requirements on input and output for procedures and functions.
8. Procedure signatures form the basis of type-checking across procedures. The name of the function, max , is intended to convey something about the semantics of the function, namely that the integer it will return is the maximum of the two integer input parameters. Ideally, the interface would specify the semantics and the requirements on parameters (for example that they must be positive integers).
9. Most programming languages do not support such facilities, however, and they are left as the task of the designer to be documented in the design documents. An exception is the Eiffel language.
10. In Ada, the unit of encapsulation is a package .A package encapsulates a set of entities such as procedures, functions, variables, and types. The package interface consists of the interfaces provided by each of those entities. The Ada package supports encapsulation by

requiring the interface of a package (called package specification) to be declared separately from the implementation of the package (called package body).

11. Figure 1 shows the Ada package specification for our dictionary unit.The package body, as can be seen in the figure, defines both the implementation of the entities defined in the package interface and the implementation of other entities internal to the module.
12. These entities are completely hidden from the clients of the package. The package specification and the package body may appear in different files and need not even be compiled together.
13. To write and compile a client module, only the service's package specification is necessary. There are significant differences between the packages of Ada and classes of C++. Even from this simple example we can see a difference between the models supported by C++ and Ada. In C++, the client can declare several instances of the dictionary class.
14. In Ada, on the other hand, a module may be declared once only and the client obtains access to only a single dictionary.

```
package Dictionary is
procedure insert (C:String; I: Integer);
function lookup(C:String): Integer;
procedure remove (C: String);
end Dictionary;
```

Fig:1

F] Separate and independent compilation:

The idea of modularity is to enable the construction of large programs out of smaller parts that are developed independently. At the implementation level, independent development of modules implies that they may be compiled and tested individually, independently of the rest of the program. This is referred to as independent compilation. The term separate compilation is used to refer to the ability to compile units individually but subject to certain ordering con-straints. For example, C supports independent compilation and Ada supports separate compilation. In Ada, as we will see later, some units may not be compiled until other units have been compiled. The ordering is imposed to allow checking of interunit references. With independent compilation, normally there is no static checking of entities imported by a module.

To illustrate this point, consider the program sketch in Figure 67, written in a hypothetical programming language. Separate compilation means that unit B ,which imports routine X from unit A , must be compiled after A . This allows any call to X issued by B to be checked statically against X 's definition in A . If the language allows module interfaces to be compiled separately from their bodies, only A 's interface must be compiled before B ; its body can be compiled at any time after its corresponding interface has been compiled.

```
Unit A
export routine X (int, int);
. . .
end A
Unit B
. . .
call X (. . .);
. . .
end B
```

FIGURE 67.Sketch of a program composed of two units

Independent or separate compilation is a necessity in the development of large programs because it allows different programmers to work concurrently on different parts of the program. It is also impractical to recompile thousands of modules when only a few modules have changed. Language concepts and features are available to allow implementations to determine the fewest number of units that must be recompiled. In general, programming languages define:

• the unit of compilation: what may be compiled independently?

• the order of compilation: are compilation units required to be compiled in any particular order?

• amount of checking between separately-compiled modules: are inter-unit interactions checked for validity?

The issue of separate compilation is at the border of the language definition
and its implementation. Clearly, if the language requires inter-unit checking
to be performed, this implies a programming environment that is able to
check module implementations against the interfaces of compilation units
from which they import services, for example a type-checking linker. Inter-
face-checking of separately compiled modules is analogous to static type-
checking for programming in the small: both are aimed at the development of
safe and reliable programs.

Libraries of modules

We have seen that C++ class and Ada's package make it possible to group
related entities into a single unit. But large programs consist of hundreds or
even thousands of such units. To control the complexity of dealing with the large number of entities exported by all these units, it is essential to be able to
organize these units into related groups. For example, it is difficult to ensure
that all the thousands of units have unique names! In general, we can always
find groupings of units that are related rather closely.

A common example of a grouping of related services is a library of modules
such as a library of matrix manipulation routines. A library collects together a
number of related and commonly used services. Clients typically need to
make use of different libraries in the same program and since libraries are
written by different people, the names in different libraries may conflict. For
example, a library for manipulating lists and a library for manipulating dictio-
naries may both export procedures named insert . Mechanisms are needed for
clients to conveniently distinguish between such identically-named services.
We have seen that the dot notation helps with this problem at the module
level. But consider trying to use two different releases of the same library at
the same time. How can you use some of the entities from one release and
some from the other? Both C++ and Ada have recent additions to the lan-
guage to deal with these issues. We will describe these facilities when we dis-
cuss specific languages: namespaces of C++ on page 295 and child libraries
of Ada on page 302.