that they are something new to learn. The best way to overcome this difficulty is to have a system that is simple in its organization and familiar in its language. The facilities of the debugging system should be organized into a few basic categories of function, which should closely reflect common user tasks. This simple organization contributes greatly to ease of training and ease of use.

The user interaction should make use of full-screen displays and windowing systems as much as possible. The primary advantage offered by such an interface is that a great deal of information can be displayed and changed easily and quickly. With menus and full-screen editors, the user has far less information to enter and remember. This can greatly contribute to the perceived friendliness of an interactive debugging system.

If the tasks a user needs to perform are reflected in the organization of menus, then the system will feel very natural to use. Menus should have titles that identify the task they help perform. Directions should precede any choices available to the user. Techniques such as indentation should be used to help separate portions of the menu. Often the most frustrating aspect of menu systems is their lack of direct routing. It should be possible to go directly to the menu that the user wants to select without having to retrace an entire hierarchy of menus.

The use of full-screen displays and techniques such as menus is highly desirable. However, a debugging system should also support interactive users when a full-screen terminal device is not present. Every action a user can take at a full-screen terminal should have an equivalent action in a linear debugging language. For example, there should be complete functional equivalence between commands and menus.

The command language should have a clear, logical, simple syntax. It should also be as similar to the programming language(s) as possible. Commands should be simple rather than compound and should require as few parameters as possible. There should be a consistent use of parameter names across the set of commands. Parameters should automatically be checked for errors in such attributes as type and range of values. Defaults should be provided for most parameters, and the user should be able to determine when such defaults have been applied.

Command formats should be as flexible as possible. The command language should minimize the use of such punctuation as parentheses, slashes, quotation marks, and other special characters. Where possible, information should be invoked through prompting techniques.

Any good interactive system should have an on-line HELP facility. Even a list of the available commands can provide valuable assistance for the inexperienced or occasional user. For more advanced users, the HELP function can be multi-level and quite specific. One powerful use of HELP with menus is to provide explanatory text for all options present on the screen. These can be

selectable by option number or name, or by filling the choice slot with a question mark. HELP should be accessible from any state of the debugging session. The more difficult the situation, the more likely it is that the user will need such information.

EXERCISES

1. Design and implement a Text editor with the following options.

   (i) create (v) delete (ix) cut

   (ii) open (vi) replace (x) paste

   (iii) save (vii) move

   (iv) insert (viii) copy

2. Design and implment a debugger.
3. Design and implement DDL, DML and DCL operations.

Chapter 8

Software Engineering Issues

This chapter contains an introduction to software engineering concepts and techniques. A full treatment of such subjects is beyond the scope of this book. Therefore, our discussion will be focused primarily on techniques that might be most useful in designing and implementing a piece of system software such as an assembler. The presentation of this material is relatively independent of the rest of the text; this chapter can be read at any time after the introduction to assemblers in Section 2.1. You may find it useful to refer to the material in this chapter as you read about the other types of system software in this book, and consider how the methods discussed could be applied in those situations as well.

Section 8.1 presents an introduction to software engineering concepts and terminology, in order to give a frame of reference for the material that follows. Section 8.2 discusses the writing of specifications to define precisely what a piece of software is to accomplish.

Section 8.3 briefly discusses one approach to *procedural* software design. We introduce data flow diagrams as a representation of the functioning of a system, and illustrate the development of a data flow diagram for a simple assembler. We then demonstrate how the data flow diagram can be used in designing the assembler as a set of relatively independent modules.

Section 8.4 introduces the *object-oriented* approach to software design. We discuss some of the central principles of object-oriented programming, and briefly indicate how these principles can be used in designing a system such an an assembler.

Finally, Section 8.5 discusses strategies for testing the individual components and the complete system.

8.1 INTRODUCTION TO SOFTWARE ENGINEERING CONCEPTS

This section contains a brief overview of software engineering terminology and ideas. Section 8.1.1 describes some of the problems that led to the development of software engineering techniques and presents several different definitions of the term *software engineering*. Section 8.1.2 discusses the

stages in the software development process and mentions some of the most important issues related to these stages. Section 8.1.3 continues this discussion into the important phase of software maintenance and evolution.

8.1.1 Background and Definitions

The development of software engineering tools and methods began in the late 1960s, largely in response to what many authors have called "the software crisis." This crisis arose from the rapid increase in the size and complexity of computer applications. Systems became much too large and complicated to be programmed by one or two people; instead, large project teams were required. In some extremely large systems, it was difficult for any one individual even to have a full intellectual grasp of the entire project. The problems in managing such a large development project led to increases in development costs and decreases in productivity. Large software systems seemed always to be delivered late, to cost more than anticipated, and to have hidden flaws. For an excellent discussion of such problems, see Brooks (1995).

We can see evidence of the continuing problems today. The purchaser of a new automobile, television set, or personal computer usually expects that the product will correctly perform its intended function. On the other hand, the first releases of a new operating system, compiler, or other software product almost always contain major "bugs" and do not work properly for some users and in some situations. The software then goes through a series of different versions or "fixes" designed to resolve these problems. Even in later releases, however, it is usual to find new flaws from time to time.

The discipline now known as *software engineering* evolved gradually in response to the problems of cost, productivity, and reliability created by increasingly large and complex software systems. Software engineering has been defined in many different ways—for example,

> the establishment and use of sound engineering principles in order to obtain, economically, software that is reliable and works efficiently on real machines (Bauer, 1972);
>
> the process of creating software systems [using] techniques that reduce high software cost and complexity while increasing reliability and modifiability (Ramamoorthy and Siyan, 1983);
>
> the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software (IEEE, 1990).

Many useful tools and techniques have been developed to help with these problems of reliability and cost. For discussions of some of these methods, see Sommerville (1996), Ng and Yeh (1990), and Lamb (1988). In spite of the advances, however, the problems are far from completely solved. Brooks (1995) gives an interesting discussion of the present state of software engineering and his view of the prospects for the future.

8.1.2 The Software Development Process

This section briefly discusses the various steps in the software development process. Figure 8.1 shows the oldest and best-known model for this process—the so-called waterfall software life-cycle model. As we shall see, this model is an oversimplification of the actual software development cycle. Nevertheless, it serves as a useful starting point for our discussions.

In the waterfall model, the software development effort is pictured as flowing through a sequenc e of different stages, much like a waterfall moving from one level to the next. In the first stage, *requirements analysis,* the task is to

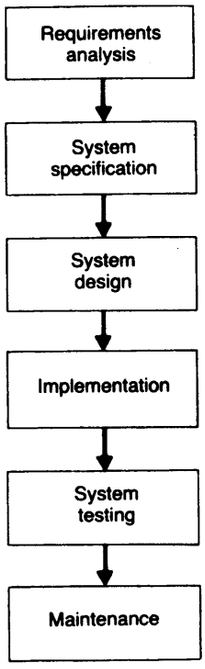


Figure 8.1 Software life cycle (waterfall model).

determine what the software system must do. The focus of this stage is on the needs of the users of the system, not on the software solution. That is, the requirements specify *what* the system must do, not *how* it will be done. In some cases, it is necessary to do much analysis and consultation with the prospective users—there are often hidden assumptions or constraints that must be made precise before the system can be constructed. The result of the requirements analysis stage is a *requirements document* that states clearly the intent, properties, and constraints of the desired system in terms that can be understood by both the end user and the system developer.

The goal of the second stage, *system specification*, is to formulate a precise description of the desired system in software development terms. The information contained in the system specification is similar to that contained in the requirements document. The focus is still on what functions the system must perform, rather than on how these functions are to be accomplished. (Many authors, in fact, consider requirements analysis and system specification to be different aspects of the same stage in the development process.) However, the approach is somewhat different. The requirements analysis step looks at the system from the point of view of the end user; the system specifications are written from the point of view of system developers and programmers, using software development terminology. Thus, the system specifications can be considered as a computer-oriented interpretation of the requirements document. We will consider examples of such specifications in Section 8.2.

The first two stages of the software development process are primarily concerned with understanding and describing the problem to be solved. The third stage, *system design*, begins to address the solution itself. The system design document outlines the most significant characteristics of the software to be developed. If a procedural approach is being taken, the system design may describe the most important data structures and algorithms to be used and specify how the software is to be divided into modules. If an object-oriented approach is taken, the system design may describe the objects defined in the system and the methods that can be invoked on each object. Smaller and less significant details are omitted—the goal is a high-level description of how the software will be constructed and how the various parts will work together to perform the desired function.

The system designer should attempt to make decisions that will minimize the problems of cost, complexity, and reliability that we have mentioned previously. In a procedural design, for example, the software might be divided into a set of relatively independent modules. An effort should be made to keep each module to a manageable size and complexity. The modular structure should also be designed so that the overall system is as easy as possible to implement, test, and modify. We will consider an example of this modular design process in Section 8.3.

After the system design is complete, the fourth step, *implementation*, can begin. In this stage of the development process, the individual modules or objects described by the design process are coded and preliminary testing is done. Although this step is what most people think of as "programming," it is actually a relatively small part of the software development effort. According to most estimates, coding should take only 10 to 20 percent of the effort involved in building a system; system design and testing consume the rest of the time (Marciniak, 1994). One of the most common mistakes made by inexperienced system developers is to begin coding too soon, before adequate planning has been done.

The actual coding of the system may use a variety of well-known techniques such as structured programming, stepwise refinement, and object-oriented programming. Good discussions of these topics, and of other methods for writing reliable software, are given by Sommerville (1996) and Ng and Yeh (1990). Thorough documentation for each part of the system is also extremely important. This documentation should include (at the least) a precise description of input and output parameters, a basic description of how the module or object works, and important details concerning any algorithms and data structures used. The style used for programming and documentation should be consistent for all of the modules of the system.

The final phase, *system testing*, is usually the most expensive and time-consuming part of the software development process. According to most estimates, testing requires 30 to 50 percent of the total development effort (Marciniak, 1994). Actually, several different levels of testing are involved. Individual parts of the system must be tested to be sure that they correctly perform their functions. The communications between the parts must be tested to be sure that they work properly together. And the entire system must be tested to ensure that it meets its overall specifications. Because the parts of the system are related to each other in a variety of ways, these types of testing overlap to some extent. In Section 8.5, we will discuss different strategies for performing these testing tasks.

The waterfall model of software development treats each stage as though it were completed before the following stage begins. In reality, this is often not the case. For example, the system specification process may reveal that certain of the requirements are incomplete or inconsistent; it is then necessary to perform more requirements analysis. During implementation or testing, flaws in the design may be discovered. Thus, there is a temporary "reverse" flow of information to earlier stages. After testing is complete, the entire process often starts over to develop a new version, or release, of the software.

In spite of its limitations, however, the waterfall model does present a rational sequence of events in the system development process. Parnas and Clements (1986) suggest that we should attempt to follow this model as

closely as possible and document the system development according to its stages, even though the actual software development may include the kind of backtracking just described.

Various authors have proposed other models of the software development process. For discussions of some of these, see Sommerville (1996).

8.1.3 Software Maintenance and Evolution

The last phase of the software life-cycle model shown in Fig. 8.1 is *maintenance.* It is tempting to believe that most of the work is done after a system has been designed, implemented, and tested. However, this is often far from true. Systems that are used over a long period of time inevitably must change in order to meet changing requirements. According to some estimates, maintenance can account for as much as two-thirds of the total cost of software (Sommerville, 1996).

Lamb (1988) identifies four major categories of software maintenance. *Corrective maintenance* fixes errors in the original system design and implementation—that is, cases in which the system does not satisfy the original requirements. *Perfective maintenance* improves the system by addressing problems that do not involve violations of the original requirements—for example, making the system more efficient or improving the user interface. *Adaptive maintenance* changes the system in order to meet changing environments and evolving user needs. *Enhancement* adds new facilities to the system that were not a part of the original requirements and were not planned for in the original design.

Maintenance can be made much easier and less costly by the presence of good documentation. The documents created during the system development (requirements, specifications, system design, and module or object documentation) should be kept throughout the lifetime of the system. It is very important to keep these documents updated as the system evolves (for example, when requirements change or new modules are added). There may be documentation that explicitly addresses questions of maintenance. For example, the designers of a system often plan for the likelihood of future change, identifying points where code or data items can easily be added or modified to meet changing requirements. Sample executions of the system should also be a part of the documentation. The test cases originally used during system development should be preserved so that these tests can be repeated on the modified version of the system.

As the system is modified, it is extremely important to maintain careful control over all of the changes being made. A software system may go through many different versions, or releases. Each such version typically involves a set of related changes to requirements, specifications, design documents, code, and user manuals. Changes to one part of the system need to be carefully

coordinated with changes to other parts. The process of controlling all of these changes to avoid confusion and interference is known as *configuration management*. For further discussions of configuration management issues, see Lamb (1988) and Sommerville (1996).

8.2 SYSTEM SPECIFICATIONS

In this section, we will examine system specifications in more detail. Section 8.2.1 discusses some of the properties that specifications should possess and examines the relationship of the specifications to other parts of the software development process. Section 8.2.2 describes several different types of specifications and gives examples of such specifications for a simple assembler. Section 8.2.3 discusses the important topic of error handling and shows how this subject is related to the system specifications.

8.2.1 Goals of System Specifications

As discussed in the preceding section, the system specification process lies between the steps of requirements analysis and system design. The requirements document is primarily concerned with the end user's view of the system and is usually written at a high level, with less important details omitted. During system specification, these details must be supplied to provide a basis for the system design to follow. The specifications must contain a complete description of all of the functions and constraints of the desired software system. They must be clearly and precisely written, must be consistent, and must contain all of the information needed to write and test the software. In order to create such specifications, the developers must examine the purpose and goals of the system more closely. The process of formulating precise system specifications often reveals areas where the requirements are incomplete or ambiguous; this requires a temporary return to the requirements analysis stage.

Although the system specifications contain more detailed information than the requirements document, they are still concerned with *what* the system must do, rather than with *how* it will be done. The selection of algorithms, data representations, etc., belongs to the following phase, system design. However, it is important that the specifications explicitly address issues such as the performance needed from the system and how the software should interact with the users. During design, it is often necessary to make choices between conflicting goals. For example, one choice of data structures might lead to more efficient processing but consume more storage space; a different choice might save space but require more processing time. The selection of one type of user interface might optimize the speed of data entry but require more initial training

time for the users. It is important that the system designers make such choices in a way that is consistent with the overall objectives of the system and the needs of the end users. Including such information in the requirements provides a basis for making the design decisions to follow.

The system specifications also form the basis for the system testing phase. Therefore, they must be written in a way that is objectively testable. In the case of "general" requirements such as efficiency and cost of operation, it is important to specify how these qualities will be measured and what will constitute acceptable performance.

As we can see, the system specifications are related closely to the other parts of the software development process. It is essential to maintain a record of these relationships so that the overall system documentation is coherent and consistent. For example, each specification should be explicitly connected with the appropriate item in the requirements document. Later in the process, design decisions may contain cross-references to the specifications that formed the basis for the decision. In the testing phase, each test case may refer explicitly to the specification that is being verified.

8.2.2 Types of Specifications

In this section, we will give examples of system specifications for an assembler. These examples are not intended to be a complete set of specifications for even a very simple assembler. Instead, they are intended to illustrate some of the possible kinds of specifications that may be written.

Figure 8.2 shows several different types of specifications. Specifications 1–6 give constraints on the input to the system—in this case, the source program. Such constraints describe the form and content of allowable inputs. A complete set of these specifications would precisely define the set of input conditions that the assembler must handle. Specification 1 deals with the *format* of the input, describing how some of the subfields of the source statement are positioned. Specification 2 gives rules for the formation of labels; such *lexical* rules determine the algorithm that must be used in scanning the input characters. (You may want to refer to the discussion of lexical analysis in Section 5.1.2.) Specification 3 describes the set of entries that are allowed to occur in a particular subfield, thus giving constraints on the *content* of the input. Specifications 4 and 5 give similar content restrictions that are also *context-dependent.* In this case, the allowable entries in a certain portion of the input (the Operand field) depend on the context in which the entry occurs—that is, the value in the Operation field of the same statement. Specification 6 describes a type of constraint commonly known as an *implementation restriction.* Such restrictions allow the system designer to define data structures that are large enough to accommodate the full range of anticipated inputs. Most

Input specifications

1. The label on a source program statement, if present, must begin in column 1 of the statement. The Operation field is separated from the Label field by at least one blank; if no label is present, the Operation field may begin anywhere after column 1.
2. Labels may be from 1 to 6 characters in length. The first character must be alphabetic (A–Z); each of the remaining characters may be alphabetic or numeric (0–9).
3. The Operation field must contain one of the SIC mnemonic opcodes, or one of the assembler directives BYTE, WORD, RESB, RESW, START, END.
4. An instruction operand may be either a symbol (which appears as a label in the program) or a hexadecimal number that represents an actual machine address. Hexadecimal numbers used in this way must begin with a leading zero (to distinguish them from symbols) and must be between 0 and 0FFFF in value.
5. A hexadecimal string operand in a BYTE directive must be of the form X'hhh...', where each h is a character that represents a hexadecimal digit (0–9 or A–F). There must be an even number of such hex digits. The maximum length of the string is 32 hex digits (representing 16 bytes of memory).
6. The source program may contain as many as 500 distinct labels.

Output specifications

7. The assembly listing should show each source program statement (including any comments), together with the current location counter value, the object code generated, and any error messages.
8. The object program should occupy no address greater than hexadecimal FFFF.
9. The object program should not be generated if any assembly errors are detected.

Quality specifications

10. The assembler should be able to process at least 50 source statements per second of compute time.
11. Experienced SIC programmers using this assembler for the first time should be able to understand at least 90 percent of all error messages without assistance.
12. The assembler should fail to process source programs correctly in no more than 0.01 percent of all executions.

Figure 8.2 Sample program specifications.

system software involves a number of such assumptions—for example, the maximum number of concurrent jobs to be run by an operating system or the maximum nesting depth of blocks in a program being compiled.

Output specifications are intended to define precisely the results to be produced by the system. Such specifications may describe the form and content of the desired output (specification 7) or the conditions under which the output is to be generated (specification 9). They may also specify constraints on the values being output (as in specification 8). Although the output values are

generally a consequence of the input, such restrictions are sometimes easier to state and check with respect to the output.

Specifications 10–12 are concerned with global characteristics of the software system, such as efficiency (specification 10), ease of use (specification 11), and reliability (specification 12). Such *quality specifications* describe overall expectations of the desired system and its operational properties. Other attributes that are often the subject of quality specifications are response time, portability, operating cost, maintainability, and training time for new users.

It is important to write quality specifications that can be objectively tested. Generalities like "efficient," "reliable," and "easy to use" may represent worthwhile goals for the system; however, without quantitative measures, it would be difficult to decide whether or not the specification had been met. In some cases, it is desirable to specify both a desired level of quality and a minimum acceptable level. For example, the desired frequency of system failures (as in specification 12) might be zero. However, it would be unrealistic to reject an entire system for a single failure. In some cases, the desired qualities for a system might be in conflict. For example, a system that is designed to be as efficient as possible might not be particularly easy to use. Features designed to make the system attractive for new users, such as detailed explanations and menus, might become cumbersome and annoying for experienced users. Therefore, it is often desirable to specify the relative importance of the various qualities desired in the system. This provides the system designer with the information needed to make intelligent choices that are in line with the overall goals for the system. Further discussions and examples of quality specifications may be found in Sommerville (1996).

Specifications often involve conditions or combinations of conditions that cannot conveniently be expressed in simple narrative sentences like those in Fig. 8.2. For example, a specification like number 4 in Fig. 8.2 might adequately describe the contents of an operand field for a SIC (standard version) program. In this case, an operand can be only a symbol or a hexadecimal constant, and issues such as relative addressing and program relocation do not arise. For a SIC/XE assembler, on the other hand, the set of conditions to be tested in processing an operand field are much more complex.

Figure 8.3 shows a sample specification for a SIC/XE assembler, expressed in *decision table* form. The upper portion of this table gives a set of *conditions*, and the lower portion describes a related set of *actions* to be taken, based on the conditions. The first column of the table lists the conditions and actions. Each column after the first describes a *rule* that specifies which actions to take under a particular combination of conditions. (You may want to refer to Section 2.2 in order to understand the logic being expressed in this specification.)

Thus, for example, Rule R5 states that if (1) the operand value type is "rel" (relative), (2) extended format is not specified in the instruction, and (3) the

Conditions/actions	Rule								
	R1	R2	R3	R4	R5	R6	R7	R8	R9
Operand value type	abs	abs	abs	abs	rel	rel	rel	rel	neither abs nor rel
Operand value < 4096?	Y	Y	N	N	—	—	—	—	—
Extended format specified?	N	Y	N	Y	N	N	N	Y	—
Operand in range for PC-relative?	—	—	—	—	Y	N	N	—	—
Operand in range for base-relative?	—	—	—	—	—	Y	N	—	—
Set bit P to	0	0		0	1	0		0	
Set bit B to	0	0		0	0	1		0	
Set bit E to	0	1		1	0	0		1	
Flag for relocation								X	
Error			X				X		X

Figure 8.3 Sample decision table specification.

operand is in range for PC-relative addressing, then the assembler should (1) set bit P to 1, (2) set bit B to 0, and (3) set bit E to 0. Rule R7 states that if (1) the operand value type is "rel", (2) extended format is not specified, (3) the operand is not in range for PC relative addressing, and (4) the operand is not in range for base-relative addressing, then an error has been detected. The entry "—" as part of a rule indicates that the corresponding condition is irrelevant or does not apply to that rule. Of course, if this decision table were used as a part of the system specification for an assembler, other specifications would be needed to define precisely what is meant by such terms as "absolute operand," "relative operand," and "in range for PC-relative addressing." Further information about the construction and use of decision tables may be found in Gilbert (1983).

8.2.3 Error Conditions

One of the most frequently overlooked aspects of software writing is the handling of error conditions. It is much easier to write a program that simply processes valid inputs properly than it is to write one that detects and

processes erroneous input as well. However, effective handling of error conditions is essential to the creation of a usable software product.

Properly written specifications implicitly define what classes of inputs are not acceptable. For example, specification 2 in Fig. 8.2 implies that a label should be considered invalid if it (1) is longer than six characters, (2) does not begin with an alphabetic character, or (3) contains any character that is not either alphabetic or numeric. Figure 8.4 shows a number of input errors derived from the input specifications in Fig. 8.2. Other erroneous input conditions may be explicitly defined by the specifications, as in the decision table in Fig. 8.3. It is extremely important that such error conditions be a part of the testing of the overall software system.

Software may take many different actions when faced with erroneous input. Sometimes a program simply aborts with a run-time error (such as a subscript out of range) or halts with no output. Obviously, these are unacceptable actions—they leave the user of the program with little or no help in finding the cause of the problem. An even worse alternative is simply to ignore the error and continue normally. This may deceive the user into thinking that everything is correct, which may lead to confusion when the output of the system is not as expected.

The preferred response to an error condition is to issue an error message and continue processing. The program should *not* terminate when an error is found, except in very unusual situations (such as running out of internal storage) when it is impossible to continue. Instead, it should process the rest of the input as completely as possible in order to detect and flag any other errors that exist. Sometimes this may involve discarding a small amount of erroneous

Error number	Violates specification	Statement		
1	1	ALPHA	LDA	BETA
2	1	ALPHALDA		BETA
3	1	LDA		BETA
4	2	ALPHAXX	LDA	BETA
5	2	1LPHA	LDA	BETA
6	2	ALP*A	LDA	BETA
7	3	ALPHA	XXX	BETA
8	4	ALPHA	LDA	7FD3
9	4	ALPHA	LDA	010000
10	5	BETA	BYTE	XA3B2
11	5	BETA	BYTE	X'01G9'
12	5	BETA	BYTE	X'A3B'

Figure 8.4 Sample input errors derived from specifications.

input (for example, skipping to the beginning of the next statement to be assembled), or taking a default action (for example, substituting 00 for an invalid operation code or address).

Effective handling of error conditions is one mark of a well-written piece of software. A program should not "crash" when presented with any input (no matter how erroneous). The philosophy "Garbage In—Garbage Out" has no place in software development. A more appropriate maxim would be "Garbage In—Meaningful Error Messages Out."

8.3 PROCEDURAL SYSTEM DESIGN

This section discusses one possible method for designing a system as a collection of procedures or modules. Section 8.3.1 introduces data flow diagrams and illustrates the development of a data flow diagram for a simple assembler. Section 8.3.2 discusses the general principles and goals of modular design—that is, what the system designer attempts to accomplish. Section 8.3.3 describes ways in which the data flow diagram can be partitioned into modules. The diagram for a simple assembler that was developed in Section 8.3.1 is used to illustrate this partitioning. Section 8.3.4 considers how the modules interact with each other and with the data objects being processed, and how these interfaces should be documented.

8.3.1 Data Flow Diagrams

A *data flow diagram* is a representation of the movement of information between storage and processing steps within a software system. The diagram shows the major *data objects* of the system, such as files, variables, and data structures. It also displays the major *processing actions* that move, create, or transform data, and the flow of data between objects and actions.

Figure 8.5(a) shows the basic notation used in a data flow diagram. Processing actions are represented by rectangular boxes and data objects by circles. Arrows show the flow of information between objects and actions. Thus, a procedure that simply copies one file to another could be represented as shown in Fig. 8.5(b). In a more complicated situation, the diagram might include the action or actions that produced File 1 and the action or actions that use information from File 2.

Figure 8.5(c) shows an action that reads a source program and produces a symbol table that contains the labels defined in the program and their associated addresses. The action may also set flags to indicate errors that were detected in the source program. Notice that the symbol table is both an input

object and an output object for this action. This reflects the fact that the action must first search the table before adding a symbol, in order to detect duplicate symbol definitions.

Data flow diagrams usually represent the most important actions and objects in a system, with less-important details omitted. For example, the action shown in Fig. 8.5(c) probably uses several other data objects, such as a location counter and working-storage variables. Likewise, the action itself could be divided into smaller actions, such as updating the location counter or scanning the source statement for a label. However, the high-level representation shown in Fig. 8.5(c) conveys the overall approach being taken.

During the system design process, data flow diagrams may initially be drawn at a relatively high level. The diagrams are then refined and made more detailed as the design progresses. As an illustration of this, let us consider a

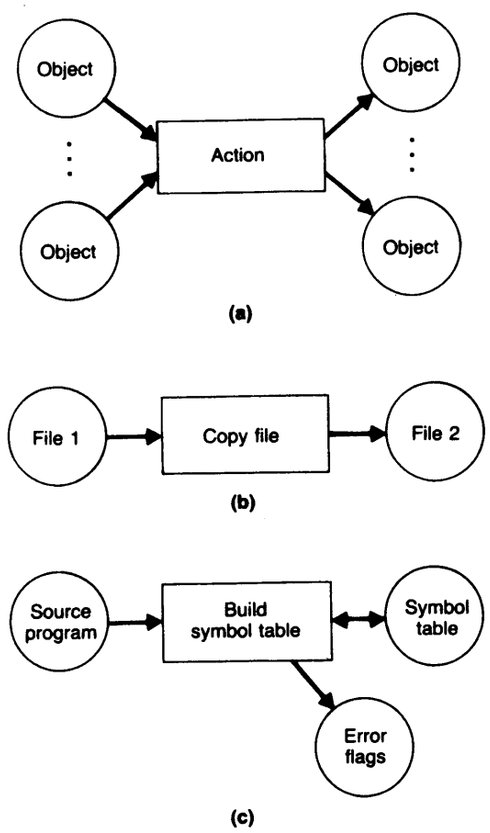


Figure 8.5 Sample data flow diagrams.

simple assembler for a SIC machine (standard version). You may want to refer to Sections 1.3.1 and 2.1 as you read this material.

Figure 8.6 shows a very-high-level data flow diagram for an assembler. There is only one action, "assemble program," and the only data objects are the source and object programs and the assembly listing. Obviously, this representation is of little value in designing the assembler. However, it may serve as a starting point for the process of developing and refining the data flow diagram.

One method for refining the diagram is to begin with the desired outputs from the system. We then add to the diagram a relatively simple action that produces each required output. (The definition of "relatively simple" obviously depends on the level of detail we wish to represent in the diagram.) If the new action does not operate directly on a primary input to the system, we must define new "intermediate" data objects to provide the required input. We then add new actions that produce these intermediate data objects, and we continue in this way until the refined diagram is complete.

Figure 8.7 illustrates this process. In Fig. 8.7(a), we have created a new action whose purpose is to format and write the object program. The object program contains all of the assembled instructions and data items, together with the addresses where they are to be loaded in memory. Thus, we define a new data object that contains this information to serve as input for the new action. At this stage, we are not concerned about how the new data object is actually produced by the system; this question will be addressed at a later step in the process.

Similarly, in Fig. 8.7(b) we have created an action to write the assembly listing. This action requires as input some of the same information that was needed for the object program. However, it also requires the source program (so that the original assembler language statements can be listed) and information about any errors that were detected during assembly. The source program is a primary

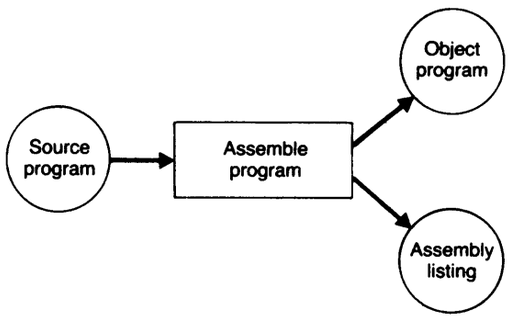


Figure 8.6 High-level data flow diagram for assembler.

input to the system, so it can be used directly by the new action. However, we must introduce a new object that contains the required error flags.

The process continues in Fig. 8.7(c). At this stage, we consider how to produce the intermediate data object that contains the assembled instructions and data with addresses. Obviously, the new action that creates this object must have the source program as one of its inputs. It also requires a table of operation codes (to translate the mnemonic instructions into machine opcodes) and a symbol table (to translate symbolic operands into machine addresses). In addition, we define a new object that contains the address assigned to each instruction and data item to be assembled. We prefer to separate the assignment of addresses from the translation process itself in order to simplify the translation action and also to make the addresses available to other actions that may be defined.

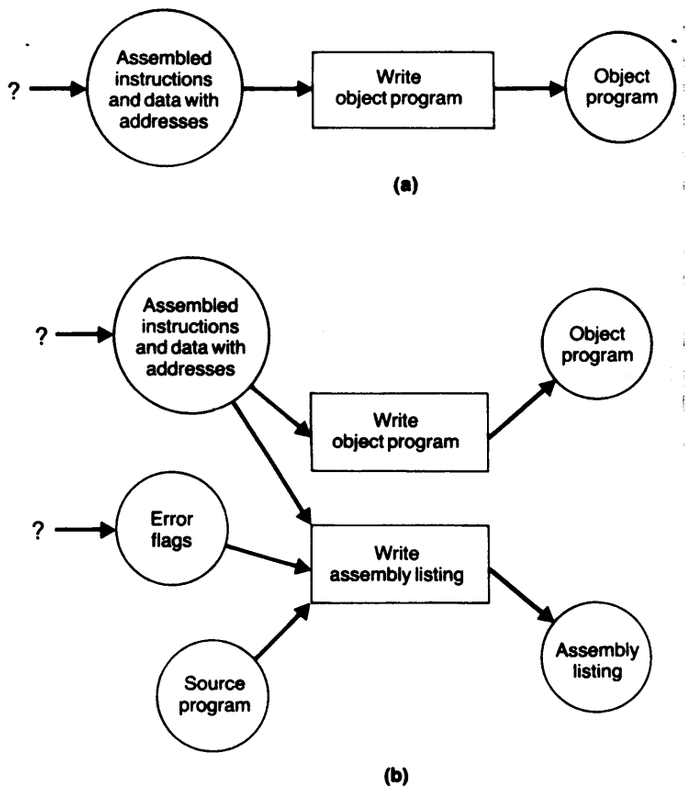
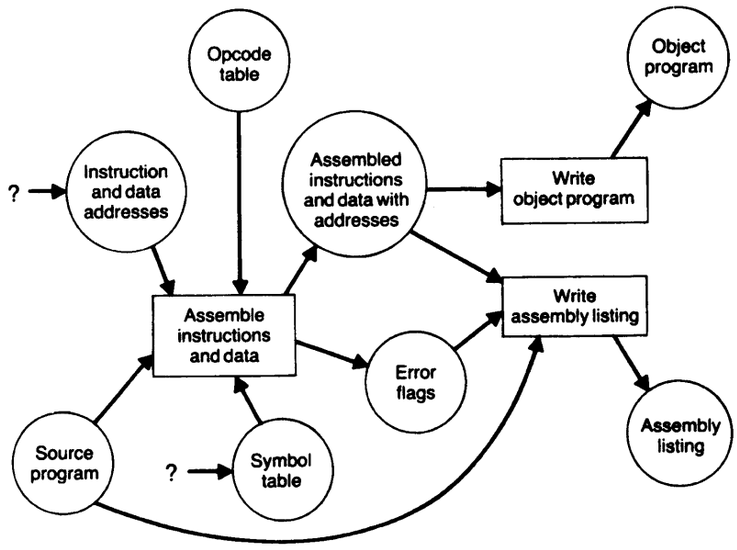
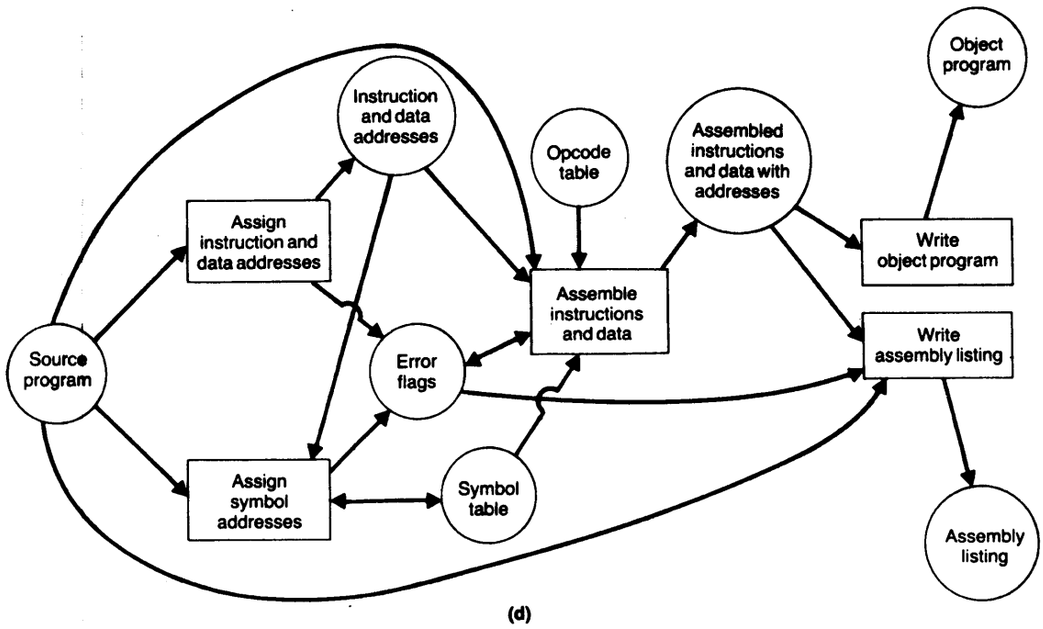


Figure 8.7 Refinement of data flow diagram for assembler.



(c)



(d)

Figure 8.7 *(cont'd)*

During the translation of instructions, certain error conditions may be detected. Thus, the data object that contains error flags is also an output from the new action in Fig. 8.7(c). (As we shall see, other actions may also set error flags in this object.) The operation code table contains only constant information that is related to the instruction set of the machine. This information may be predefined as part of the assembler itself, instead of being produced during each assembly of a source program. Thus, the operation code table may be treated as though it were a primary input to the system—we will not need to create any new action to produce this data object.

Figure 8.7(d) shows the final step in the development of the data flow diagram for our simple assembler. We have created one new action to compute the addresses for the instructions and data items for the program being assembled. This action operates by scanning the source program and noting the length of each instruction and data item, as described in Section 2.1. Another new action uses these addresses to make entries in the symbol table for each label in the source program. Both of the new actions may detect errors in the source program, so they may need to write into the data object that is used to store error flags. With these new actions defined, there are no "disconnected" actions or objects. Thus, the data flow diagram in Fig. 8.7(d) is complete.

The data flow diagram is intended to represent the flow and transformation of information from the primary inputs through intermediate data objects to the final outputs of the system. As the diagram is developed, it is important to write down documentation for the data objects and processing actions that are being defined. For example, the documentation should describe what data is stored in each object and how each action transforms the data with which it deals. However, the data structures being used to store the information and the algorithms used to access this information are not a part of the data flow representation. Likewise, the mechanisms by which data is passed from one processing action to another are not specified by the representation. Such implementation details are a part of the modular design process that we discuss in the following section. Thus, the data flow diagram and associated documentation can be considered as an intermediate step between the specifications (which describe *what* is to be done) and the system design (which describes *how* the tasks are to be accomplished).

8.3.2 General Principles of Modular Design

The data flow diagram for a system represents the flow and transformation of information from the primary inputs through intermediate data objects to the final outputs of the system. However, there are many different ways in which these flows and transformations could be implemented in a piece of software.

For example, consider the action in Fig. 8.7(d) that assigns instruction and data addresses. This action might be implemented as a separate pass over the source program, computing all addresses before any other processing is done. In that case, the object that contains the addresses might be a data structure with one entry for each line of the source program. This structure would then be used by the other actions of the assembler.

On the other hand, the action that assigns addresses might deal with the source program one line at a time. It might compute the address for each instruction or data item and then pass this address to the other parts of the assembler that require it. In that case, the data object created might be a simple variable, containing only the address for the line currently being processed.

Similar options exist for many of the other actions and data objects in Fig. 8.7(d). Thus, this data flow diagram could describe an ordinary two-pass assembler. However, it could equally well describe a one-pass assembler or a multi-pass assembler (see Section 2.4). The choices between such alternatives are made by the system designer as the data flow diagram is converted into a modular design for a piece of software.

Obviously, the goal of the modular design process is a software design that meets the system specifications. However, it is almost equally important to design systems that are easy to implement, understand, and maintain. Thus, for example, the modules should be small enough that each could be implemented by one person within a relatively short time period. Modules should have high *cohesion*—that is, the functions and objects within a module should be closely related to each other. At the same time, the modules in the system should have low *coupling*—that is, each module should be as independent as possible of the others. Systems organized into modules according to these "divide and conquer" principles tend to be much easier to understand. They are also easier to implement, because a programmer needs to understand and remember fewer details in order to code each module. The resulting system is easier to maintain and modify, because the changes that need to be made are usually isolated within one or two modules, not distributed throughout the entire system. In the remainder of this section, we will see examples of the application of these general principles to the design of an assembler.

8.3.3 Partitioning the Data Flow Diagram

The modular design process may be thought of as a partitioning of the data flow diagram into modules. As the modules are defined, the interfaces between modules and the data structures to be used for objects are also specified. One common approach to this partitioning, called *top-down design*, begins by dividing the data flow diagram into a relatively small set of major processing units.

Each such unit may then be divided into subunits, and the process continues until the design is complete.

The division of the diagram (or a portion of the diagram) into units may be based on a variety of factors. Two of the most common criteria are the sequence in which functions are performed and the type of function being performed. For example, Fig. 8.8 shows the data flow diagram for our simple assembler partitioned into two passes. This division is based on the processing sequence to be used—assigning addresses to labels in Pass 1 and then assembling the source statements in Pass 2. (You may want to review the discussion in Section 2.1 for further explanation of this division into passes.) At a later stage in the design process, we might use divisions based on type of function. For example, the task of assembling a single line from the source program might be divided into one module that assembles machine instructions, one module that assembles data constants, and so on.

Another important factor to consider in modular design is the desirability of minimizing the coupling between modules. Consider, for example, the portion of the data flow diagram that is shown in Fig. 8.9(a). Suppose that each of the two actions shown is implemented as a separate module. These two modules access the symbol table directly (to add new entries and to search the table).

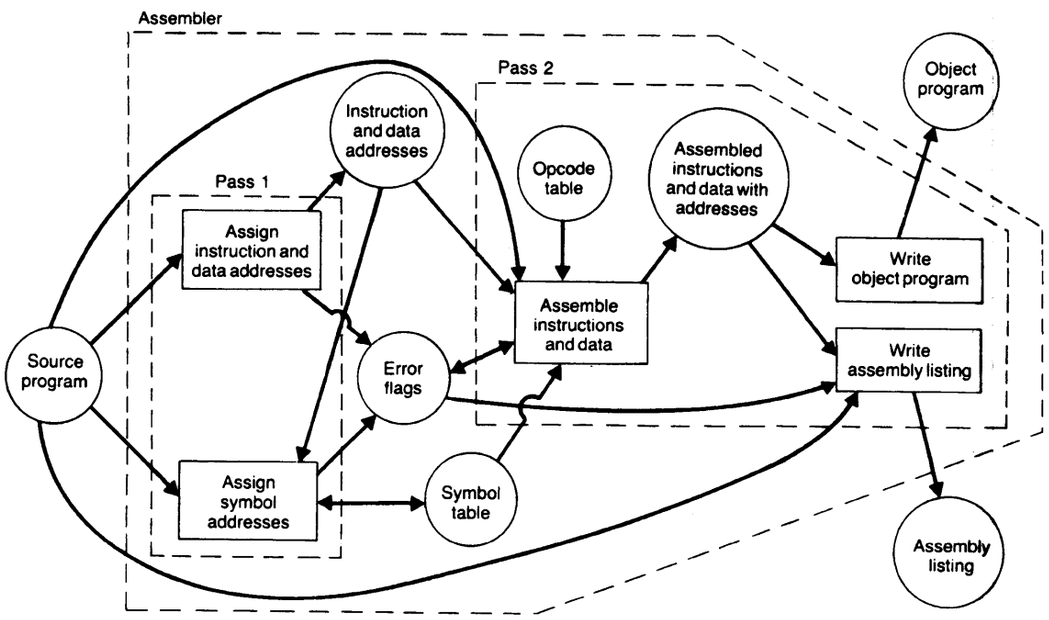


Figure 8.8 Division of assembler data flow diagram into passes.

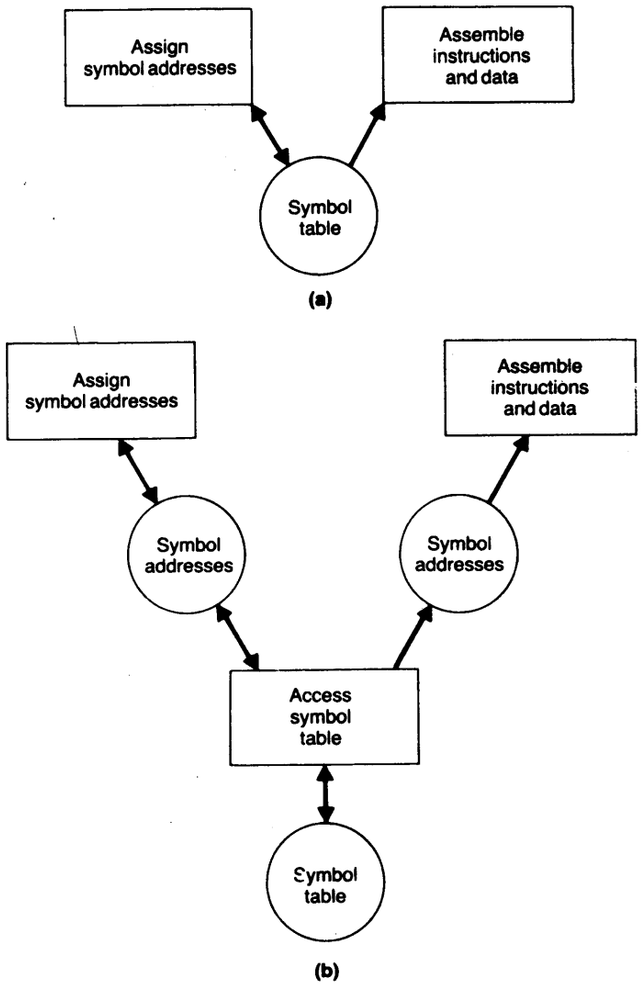


Figure 8.9 Isolation of symbol table design.

Thus, both modules must know the internal structure of the symbol table. For example, if the symbol table is a hash table, then both modules must know the size of the table, the hashing function, and the methods used for resolving collisions. This creates more work for the programmers who implement the modules. It also leads to duplication of effort, because the same code is written twice, and it creates additional possibilities for errors in the implementation.

Similar problems may occur during the maintenance phase. If the organization of the table or the methods for accessing it are changed, then both of

the processing modules must be modified. As before, this requires more work and may lead to errors if one module is updated and the other is not.

The difficulties just described are a consequence of the undesirable coupling between the two modules—closely related items of information and processing logic occur in both modules. The modules also exhibit relatively poor cohesion—each module must contain information and logic that is related to the design of the symbol table, instead of focusing only on the logical requirements of the specific processing task being performed.

A better design, with increased cohesion and reduced coupling, is shown in Fig. 8.9(b). We have defined a new module whose sole purpose is to access the symbol table. This module is called by the other two whenever they need to perform any operation on the table. Thus, the two original modules need only know the calling interface (parameters, etc.) used to invoke the "access" module. The internal structure of the symbol table—size, organization, algorithms for access, etc.—are of concern only within the new module that performs the actual access. This reduces the amount of knowledge that must be included in the two main processing modules. It also simplifies the maintenance of the system in case the internal details of the table structure need to be changed.

In the process just illustrated, the effect of a design decision (i.e., the internal structure and representation of the symbol table) was "isolated" within a single module. This design principle is sometimes referred to as *isolation of design factors,* or simply *factor isolation.* The same general concept is also often called *information hiding* (because a module "hides" some design decision), or *data abstraction* (because the rest of the system deals with the data as an "abstract" entity, separated from its actual representation). Further discussions of these topics can be found in Ng and Yeh (1990) and Lamb (1988).

Figure 8.10 shows a modularization of the data flow diagram from Fig. 8.8 according to the principles just described. This design includes the new module introduced in Fig. 8.9 (Access_symtab); there are also several other similar changes. The source statements, instruction and data addresses, and error flags that are communicated from Pass 1 to Pass 2 are included in an intermediate file. (A discussion of the reasons for this design decision may be found in Section 2.1.) A new module (Access_int_file) has been defined to handle all of the reading and writing of this intermediate file. The reasons for including this module are essentially the same as those discussed above—all of the details concerning the structure and access techniques for the intermediate file are isolated within a single module and removed from the rest of the system.

Likewise, we have defined a module (P2_search_optab) whose sole purpose is to access the operation code table. This design decision is somewhat different from the two just discussed, because it does not materially reduce the coupling between modules. (Whether or not the new module is defined, there is still only one place in the system where the structure of the table must be known.)

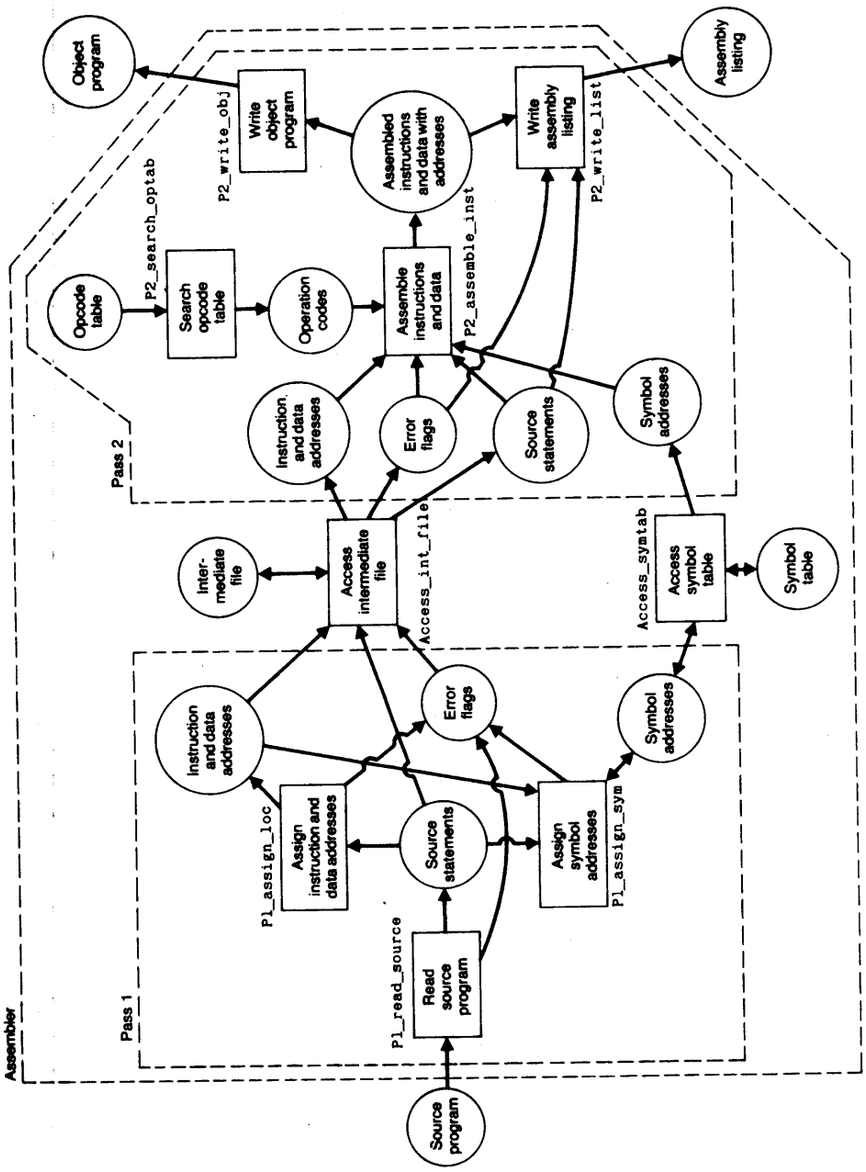


Figure 8.10 Modularized assembler design.

However, the decision does make the module that assembles instructions smaller and less complex. It also improves module cohesion by separating two logically unrelated functions that were previously part of the same module. Thus, it leads to a modular structure that is easier to implement, understand, and modify. For similar reasons, we have introduced a module (P1_read_source) that reads the source program and passes the source statements to the rest of the assembler. This module could, for example, handle details such as scanning for the various subfields in a free-format source program.

Figure 8.10 represents one possible stage in the modular design of our assembler. Depending upon the detailed specifications for the assembler, however, some of these modules may still be larger and more complex than is desirable. In that case, the decomposition could be carried further. For example, module P2_assemble_inst could be divided into several submodules according to the type of statement being processed—one to assemble Format 3 instructions, one to process BYTE assembler directives, etc. There may also be a number of "common" or "utility" functions that are used by more than one module in the system. These functions might include, for example, conversion of numeric values between internal (integer) and external (hexadecimal character string) representations. Each such function could be isolated within its own module and called by the other modules as needed. This isolation would improve module cohesion and reduce module coupling, resulting in the benefits previously described.

8.3.4 Module Interfaces

Figure 8.10 describes the decomposition of a problem into a set of modules. However, it does not specify the sequence in which these modules are to be executed or the interfaces between the modules. These are questions that must be addressed by the system designer before the implementation can begin.

There are often many ways in which a given set of modules can be organized into a system. For example, consider the three modules that make up Pass 1 in Fig. 8.10. In one possible organization, module P1_read_source would be called by the main procedure for Pass 1. For each call, P1_read_source would read a line from the source program. It would then call P1_assign_loc (for noncomment lines) to assign an address to the current statement. For statements containing a label, P1_read_source would call P1_assign_sym to make the required entry in the symbol table. A similar organization would have P1_read_source call P1_assign_loc for every noncomment line read. After calculating the appropriate address, P1_assign_loc would then call P1_assign_sym itself (instead of returning immediately to P1_read_source).

In the organization just described, the processing of Pass 1 is "driven" by P1_read_source. On the other hand, the processing could also be controlled by

P1_assign_loc. This module could call P1_read_source whenever it needs another line from the source program. It could assign an address to the line returned by P1_read_source and then call P1_assign_sym. Similarly, the processing could be driven by P1_assign_sym.

Yet another possibility is for the main procedure of Pass_1 to call all three of the other modules directly. Thus, Pass_1 could call P1_read_source to read each line from the source program. Pass_1 would then call P1_assign_loc and P1_assign_sym in turn, passing as a parameter the source line just read. Because of its simplicity, we have chosen this form of organization for the remainder of our discussions. We have also chosen to follow a similar organization in Pass 2. Figure 8.11(a) summarizes this calling structure for the modules of our assembler. In this diagram, an arrow from one module to another indicates that the first module may call the second.

A closely related issue is the placement of data objects within the modular structure. If an object is used by only one module, it is natural to place that object within the module that uses it. This is especially true in the case of modules whose purpose is to hide the internal structure of the object. For example, the symbol table in our assembler design should probably be declared within module Access_symtab. If an object is required by more than one module, the data can be shared either via parameter passing or through the use of global variables. In the first approach, the data object itself is located within one module, which passes the information in the form of parameters to other modules as needed. For example, the data object that contains the source statement currently being processed could be declared in the module Pass_1 and passed as a parameter to P1_read_source, P1_assign_loc, and P1_assign_sym. In the second approach, the data object is made global or common to the modules that require it. For example, suppose that the modules P1_read_source, P1_assign_loc, and P1_assign_sym are contained within the module Pass_1. If the data object containing the source statement were also declared within Pass_1, it could be directly accessible to the other three modules, without the need for parameter passing.

Although these two approaches to data placement—parameter passing and global variables—can be viewed as equivalent in their effect, the choice of one over the other may be influenced by a variety of factors. Some of these factors are related to the programming language being used—for example, the amount of overhead involved in parameter passing and the mechanisms for allocating and using local and global variables. Other factors are related to the structure of the software system itself. The use of parameters provides a clearly defined and limited interface between modules. However, it may be inconvenient if the need for data transmission does not closely match the calling structure of the system. For example, a particular item of information might have to be passed through a chain of calls before reaching the module that requires it. On the other hand, the

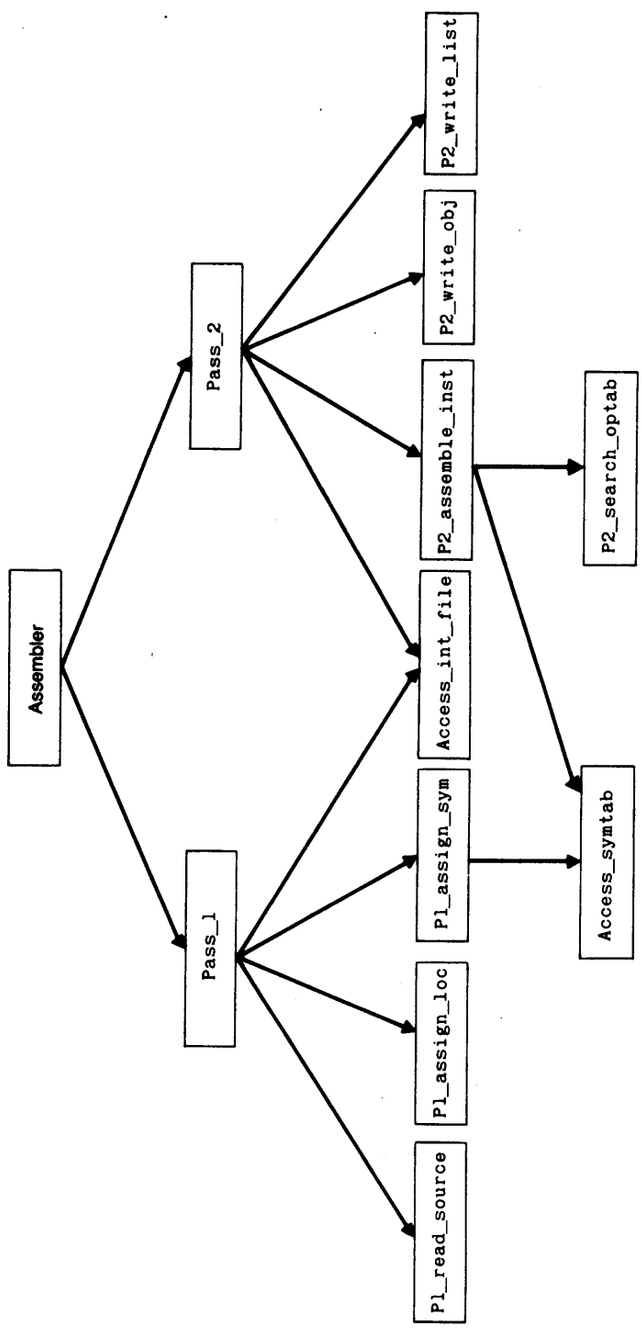


Figure 8.11(a) Calling structure for modules of assembler.

Module	Parameters	Called by	Calls
Assembler	—	User	Pass_1 (once) Pass_2 (once)
Pass_1	—	Assembler	P1_read_source (for each source line) P1_assign_sym (for each noncomment line) P1_assign_loc (for each noncomment line) Access_int_file (for each source line)
Pass_2	—	Assembler	Access_int_file (for each source line) P2_assemble_inst (for each noncomment line) P2_write_obj (for each noncomment line) P2_write_list (for each source line)
Access_int_file	Request code (I) Return code (O) Source statement (I/O) Current location counter (I/O) Error flags (I/O)	Pass_1 Pass_2	—
Access_symtab	Request code (I) Return code (O) Symbol(I) Address (I/O)	P1_assign_sym P2_assemble_inst	—
P1_read_source	Return code (O) Source statement (O) Error flags (O)	Pass_1	—
P1_assign_loc	Source statement (I) Error flags (O) Current location counter (I) Next location counter (O)	Pass_1	—
P1_assign_sym	Source statement (I) Error flags (O) Current location counter (I)	Pass_1	Access_symtab (for each label)
P2_assemble_inst	Source statement (I) Error flags (I/O) Current location counter (I) Object code (O)	Pass_2	P2_search_optab (for each instruction) Access_symtab (for each operand symbol)
P2_search_optab	Mnemonic opcode (I) Return code (O) Machine opcode (O)	P2_assemble_inst	—
P2_write_obj	Current location counter (I) Object code (I)	Pass_2	—
P2_write_list	Source statement (I) Current location counter (I) Error flags (I) Object code (I)	Pass_2	—

(b)

Figure 8.11(b) Parameters and calling sequence for modules of assembler.

use of global variables provides a simple and efficient means of sharing data. However, it can increase the coupling between modules, because any module in the entire system could potentially use or modify any global data object.

In general, it seems desirable to avoid the use of global variables unless there is a clear reason for preferring to use them in a specific situation. If global variables are used, it is important to document clearly the use of these variables in all of the modules affected. Further discussions of module interfaces and the implications of data placement may be found in Gilbert (1983).

Figure 8.11(b) shows a high-level description of the calling sequence and parameters for our modular assembler design. In this description, we have assumed that all data values are passed between modules in the form of parameters. This type of documentation is an extremely important part of the design process, because it forms the basis for the implementation phase to follow. An actual system design, of course, would include much more detailed information than is present in this example. The parameters for each module would be described completely, with data types specified and detailed descriptions of the contents and use of each parameter. The processing to be performed by each module would also be described carefully and precisely. You may want to write down some of these details for yourself, in order to gain further insight into the process of specifying a software design.

8.4 OBJECT-ORIENTED SYSTEM DESIGN

This section introduces a modern alternative to the procedural system design methods discussed in Section 8.3. *Object-oriented design* focuses on the objects handled by the system, rather than on algorithms. Programs are designed and implemented as collections of objects, not as collections of procedures.

The object-oriented paradigm has become increasing popular in recent years. Some people had hoped that this approach would be the "silver bullet" to cure all of the problems of software engineering (Brooks, 1995). Although it still appears that there is no universal "best" design method, the object-oriented strategy has been found to have a number of potential advantages.

Section 8.4.1 introduces the fundamental principles and concepts of object-oriented programming. Section 8.4.2 illustrates the use of object-oriented methods as applied to the design of an assembler.

8.4.1 Principles of Object-Oriented Programming

In the *object-oriented programming* (OOP) methodology, programs are structured as collections of objects, not as collections of procedures. An *object* contains some data and defines a set of operations on that data that can be invoked by

other parts of the program. Objects may represent physical things that exist in the real world, or they may represent software entities such as data structures, user-interface menus, and memory managers.

The data contained by an object consists of the values of its *instance variables*. These instance variables are normally inaccessible from outside the object. Other parts of the program can view or manipulate the data stored in the object only by *invoking* one of the operations defined by the object. These operations are usually referred to as *methods;* depending on the programming language, they may also be called *operations* or *member functions*. Invoking a method on an object is usually accomplished by sending a message to the object. Depending on the particular method being invoked, a reply message may also be returned to the invoker.

Consider, for example, an object that represents the symbol table used by an assembler. The methods defined by this object might include operations such as Insert_symbol and Lookup_symbol. The instance variables of the object would be the contents of the hash table (or other data structure) used to store the symbols and their addresses. The representation of the instance variables—for example, the way the hash table is organized—would be invisible to the rest of the assembler.

Compare this approach with the isolation of design factors that we discussed in Section 8.3.3. In effect, the object that represents the symbol table combines the "symbol table" data structure and the "access symbol table" module that are shown in Fig. 8.9(b). The object-oriented representation provides the same advantages of data abstraction and information hiding that we discussed in Section 8.3.3. For example, any changes in the internal organization of data in the object do not affect the rest of the assembler. The OOP term for this kind of abstraction is *encapsulation*.

However, there is much more to OOP than encapsulation. In the object-oriented paradigm, each object is created as an *instance* of some class. A *class* can be thought of as a template that defines the instance variables and methods of an object. It is possible to create many objects from the same class. For example, suppose that an assembler is designed to translate programs for different versions of the target machine (such as SIC and SIC/XE). The assembler might make use of a class named Opcode_table. A separate instance of this class (i.e., a separate object) could be created to define the instruction set for each version of the target machine.

Classes can be related to each other in a variety of ways. Consider, for example, Fig. 8.12(a). An object of the class Source_program could be used to represent an assembler language program. This object might contain a variety of information about the program itself—for example, the total program size and an indication of whether or not errors have been detected. It could also include a collection of objects of the class Source_line. Each of these objects would represent a single line of the program.

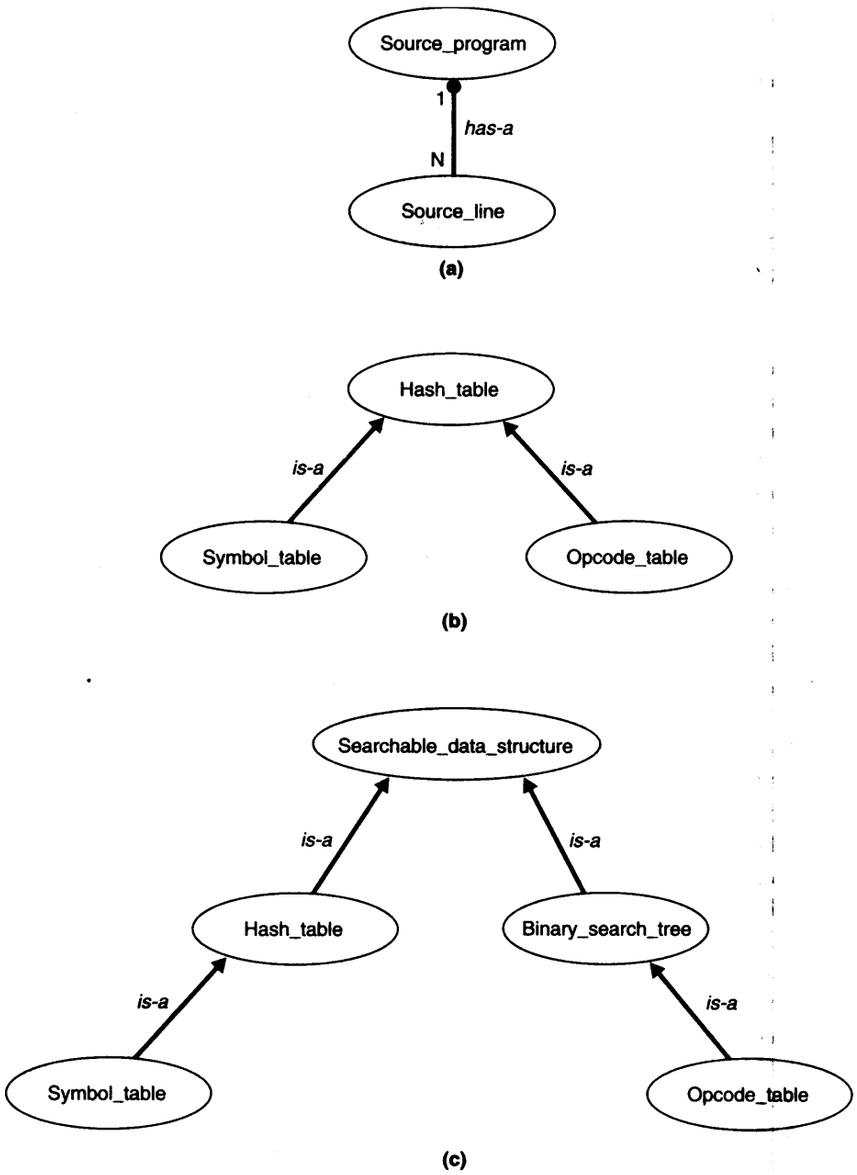


Figure 8.12 Examples of relationships between classes.

In this example, the relationship between the class Source_program and the class Source_line is one of inclusion or aggregation. In OOP terms, this is called a "has-a" relationship. The diagram indicates that there is a 1:N relationship between one instance of Source_program and many instances of Source_line.

It is also possible for one class to be a *subclass* of another. For example, Fig. 8.12(b) shows a class Symbol_table and a class Opcode_table. These are both subclasses of the *base class* Hash_table. In OOP terms, this is called an "is-a" relationship.

Subclassing is very important in the object-oriented paradigm. When a subclass is created, it automatically *inherits* all of the instance variables and methods of the base class. For example, suppose the class Hash_table defines methods called Insert_item and Search_for_item. When the classes Symbol_table and Opcode_table are declared, they automatically contain definitions of these same methods. Likewise, they automatically incorporate whatever mechanism is used in the class Hash_table for organizing and accessing the data (instance variables).

The instance variables and methods inherited by a subclass can be overridden to add new or specialized behavior to the subclass. For example, the instance variables of the class Symbol_table could be changed to include symbol addresses, error flags, and other needed information. Likewise, the instance variables of Opcode_table could be changed to include information about instruction formats. The method Insert_item could be deleted from the subclass Opcode_table to prevent accidental changes to the contents of this table.

Subclassing and inheritance allow the programmer to reuse existing code that has already been developed and tested for a more generic case. The symbol and opcode tables can be implemented without needing to recode the mechanics of managing the hash table (hashing function, collision resolution method, etc.) The only things that need to be written are the parts that are specific to each subclass. This reuse of existing code can result in significant savings of time and effort in design, implementation, and testing.

Figure 8.12(c) illustrates another way of using subclasses. In this example, there is a *superclass* named Searchable_data_structure, which defines the methods Insert_item and Search_for_item. The classes Hash_table and Binary_search_tree are subclasses of Searchable_data_structure. Thus they inherit these two methods. The implementations of the methods are different in the classes Hash_table and Binary_search_tree, because different underlying data representations are used. However, the names of the methods and the way they are invoked are the same for both subclasses.

Figure 8.12(c) indicates that Symbol_table is a subclass of Hash_table and Opcode_table is a subclass of Binary_search_tree. If the method Search_for_item is invoked on an instance of Symbol_table, it will result in a retrieval from a hash table. If the same method is invoked on an instance of Opcode_table, it will result

in a search of a binary tree. This is an example of *polymorphism*—the same message (method invocation) sent to different objects can result in different behavior.

Polymorphism is one of the most powerful features of the object-oriented programming methodology. Suppose, for example, we decide to change Symbol_table from a hash table to a binary search tree (or to some other subclass of Searchable_data_structure). Because of the polymorphism, this change will have absolutely no effect on the rest of the assembler.

Encapsulation, inheritance, and polymorphism are the key characteristics of the object-oriented paradigm. To be considered truly object-oriented, a programming language must support at least these three features.

8.4.2 Object-Oriented Design of an Assembler

In this section, we illustrate the use of object-oriented methods as applied to the design of an assembler. Object-oriented design can be viewed in different ways. Booch (1994) suggests that there are really two different development processes, which he calls *micro* and *macro*.

Booch's macro process represents the overall activities of the development team on a long-range scale (perhaps many months at a time). It includes the following activities:

1. Establish the requirements for the software (conceptualization).
2. Develop an overall model of the system's behavior (analysis).
3. Create an architecture for the implementation (design).
4. Develop the implementation through successive refinements (evolution).
5. Manage the continued evolution of a delivered system (maintenance).

This macro process repeats itself after each major release of a software product. The overall sequence of events is similar to the waterfall model depicted in Fig. 8.1. However, Booch argues that object-oriented development is inherently iterative, so that reverse flows of information like those discussed in Section 8.1.2 are inevitable.

Booch's micro process essentially represents the daily activities of the system developers. It consists of the following activities:

1. Identify the classes and objects of the system.
2. Establish the behavior and other attributes of the classes and objects—for example, the methods that are applicable to each.
3. Analyze the relationships among the classes and objects—for example, the use of aggregation, inheritance and polymorphism.

4. Specify the implementation of the classes and objects—for example, the data structures and algorithms that are used in each.

These activities may be repeated as needed, with increasing levels of detail being considered.

We will focus on this micro process at a relatively high level, similar to the discussions of procedural design in the previous section. It is natural to begin by identifying software objects that correspond to real-world objects—in this case, the source program, the object program, and the assembly listing.

During assembly, we will need to perform several different operations on individual lines of the source program. Therefore, we choose to make each source line a separate object. The source program contains the objects that represent the individual source lines, as shown in Fig. 8.12(a). On the other hand, the lines in the assembly listing are not manipulated separately, except for being inserted into the listing itself. Thus we choose to represent these lines as instance variables within the listing object, not as separate objects themselves. A similar reasoning applies to the individual parts of the object program. Of course, all of these choices are design decisions, which could be made in different ways.

Our knowledge of the way an assembler operates also leads us to identify two other objects—a symbol table and an opcode table. These objects will be used as described in Section 2.1. We assume that the symbol table and the opcode table will be derived from a superclass of searchable data structures, as shown in Fig. 8.12(c).

Figure 8.13 lists the objects we have identified so far. It also briefly indicates what each object contains and what methods can be invoked on the object. Clearly, these specifications must be made more precise before any implementation is attempted. However, they are sufficient to guide us at this level of the design process.

Now we are ready to look at how the objects of the assembler interact with each other. This can be done in several different ways. Figure 8.14 shows an *object diagram* which indicates the methods that are invoked by each object. Thus, for example, the object Source_program invokes the methods Create, Assign_location, and Translate on the Source_line objects. The object Source_line invokes the methods Search and Enter on the object Symbol_table, as well as methods on several other objects.

It is possible to include several other kinds of information in an object diagram. For example, the diagram may also indicate the class of each object. The invocations may be numbered to indicate the sequence in which they occur, and the flows of information they cause. Different types of annotations can be used to describe the synchronization among the objects and the degree of "visibility" between objects. Discussions of such issues can be found in Booch (1994).

Source_program

Contents

The program to be translated. Includes the current location counter value and a summary of errors detected in the program. Also includes one object of class Source_line for each line of the program.

Methods

Assemble	—	Translate the source program, producing an object program and an assembly listing.

Source_line

Contents

A line of the source program. Includes the location counter value and indications of errors detected for the line.

Methods

Create	—	Create and initialize a new instance of Source_line.
Assign_location	—	Assign a location counter value to the line; return an updated location counter value to the invoker. Enter the label on the line (if any) into the symbol table.
Translate	—	Translate the instruction or data definition on the line into machine language. Make entries in the object program and assembly listing.
Record_error	—	Record an error detected for the line.

Symbol_table

Contents

Labels defined in the source program, with the location counter value assigned to each.

Figure 8.13 Preliminary specification of objects for assembler.

Methods

Enter	—	Enter a label and location counter value into the table. Return an error indication if the label is already defined.
Search	—	Search the table for a specified label. Return the location counter value associated with the label, or an error indication if the label is not defined.

Opcode_table

Contents

Mnemonic instructions to be recognized by the assembler. Includes the machine instruction format and opcode to be used in assembling each instruction, and a description of requirements for operands.

Methods

Search	—	Search the table for a specified mnemonic instruction. Return information about the instruction format and operands required, or an error indication if the mnemonic instruction is not defined.

Object_program

Contents

The object program resulting from the assembly. Includes the machine language translation of instructions and data definitions from the object program. Also includes an indication of program length.

Methods

Enter_text	—	Enter the machine language translation of an instruction or data definition into the object program.
Complete	—	Enter the program length and complete the generation of the external object program file.

Figure 8.13 (*cont'd*)

Assembly_listing

Contents

A listing of the lines of the source program and the corresponding machine language translation for each line. Also includes a description of errors detected for each line, and a summary of errors detected in the program.

Methods

Enter_line	—	Enter a source line, the corresponding machine language translation, and a description of errors detected for the line into the assembly listing.
Complete	—	Enter a summary of errors detected and complete the generation of the external assembly listing file.

Figure 8.13 (*cont'd*)

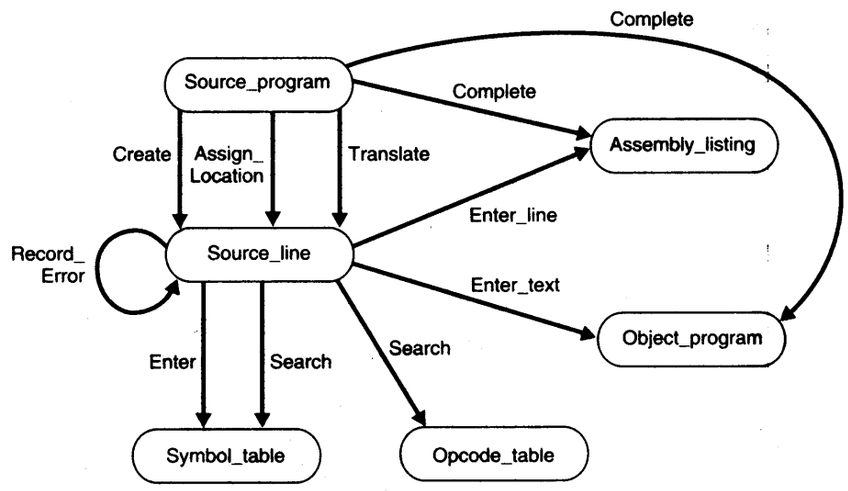


Figure 8.14 Object diagram for assembler.

In some situations, it is useful to represent the system in a different way. Figure 8.15 shows an *interaction diagram* for our preliminary assembler design. Interaction diagrams make it easier to visualize the sequence of object invocations, and the flow of control between objects. Annotations can be used to indicate iteration and conditions that affect the interactions between objects.

In an interaction diagram, each object is represented by a dashed vertical line. The invocation of a method is shown by a horizontal line between one object and another. The sequence of operations is indicated by their vertical position in the diagram. A *script* is often written at the left-hand side of the diagram to describe conditions and iteration. A narrow vertical box can be used to indicate the time that the flow of control is focused in each object.

Consider, for example, the interactions shown in Fig. 8.15. The primary focus of control is in the object Source_program. The Assemble method of Source_program begins by creating a new instance of Source_line for each line in the input file for the assembler. The method Assign_location is then invoked for each Source_line object. During its period of control, Source_line may invoke Enter to make an entry in the symbol table. If errors are detected, it may invoke the method Record_error on itself.

After the iteration of Assign_location is complete, a second iteration is used to translate each source line. During this process, Source_line may invoke methods on several other objects, depending on the conditions encountered. You are encouraged to trace through the flow of invocations described in

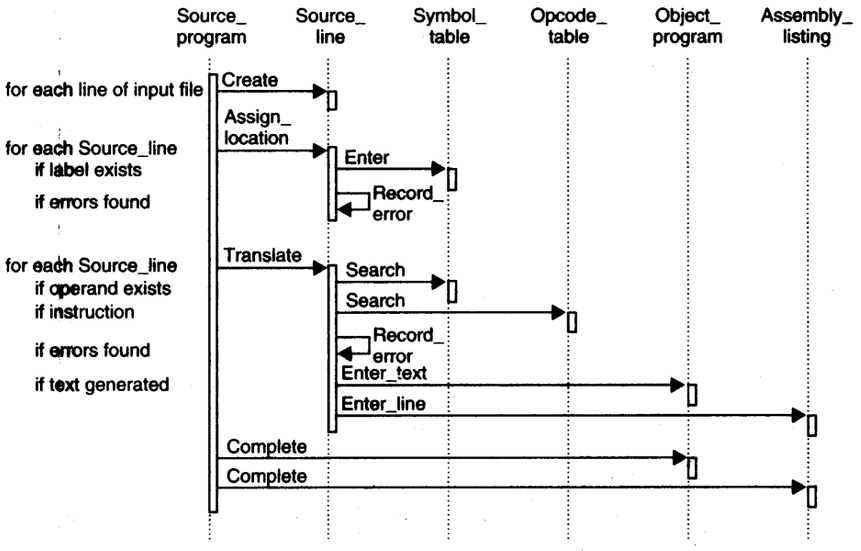


Figure 8.15 Interaction diagram for assembler.

Fig. 8.15. You may also want to compare this diagram with the algorithmic representations in Fig. 2.4.

Notice that all of the system descriptions we have discussed focus on the *behavior* of objects, not on how the objects are represented. Object-oriented design delays decisions about representation of objects until as late as possible. This helps avoid premature implementation decisions that might bias the design process.

When implementation details are considered, the first step is to examine existing class libraries. It may be that an existing class already implements some of the behavior that is needed in a particular object. In this case, the designer can make use of a *derived class* (i.e., a subclass of the existing class). The behavior of the existing base class will automatically be inherited by the new derived class. Only the parts of the behavior that are unique to the new system will need to be implemented from scratch.

Finally, each object is implemented using an appropriate object-oriented language. Representations for the instance variables are chosen, and algorithms for each method are selected. The details of implementation depend to some extent on the language being used, and are beyond the scope of this chapter. Discussions of implementation issues, and more information about object-oriented programming and design, can be found in Booch (1994).

8.5 SYSTEM TESTING STRATEGIES

This section provides an introduction to system testing techniques. Many books and papers have been written on software testing methodologies; a detailed discussion of such topics is beyond the scope of this chapter. Instead, we consider general strategies that can be used in testing a set of related modules that make up a software system. These modules may be classes or objects that are defined by an object-oriented design process, or they may be procedures that are defined by a procedural design process. We focus on alternatives for the sequence in which the modules of a system are coded and tested. Further information about software testing in general can be found in Sommerville (1996).

Section 8.5.1 introduces some terminology and briefly mentions the various levels of system testing. Sections 8.5.2 and 8.5.3 discuss two commonly used sequences for testing a collection of modules.

8.5.1 Levels of Testing

A large software system, composed of many thousands of lines of source code, is almost always too complex to be debugged effectively all at once—when errors occur, there are too many places to look for them. Most such systems

are tested in a series of stages. At each stage, the amount of code actually being debugged is kept relatively small, so that it can be tested thoroughly and efficiently.

In the *unit test* phase, individual modules are tested in isolation from the rest of the system. During this process, it is usually necessary to simulate the interaction of the module being tested with the other parts of the system. Methods for doing this are discussed in the following sections.

The unit test of an individual module is usually governed by the module specifications. That is, test cases are generated from the specifications for the module, without considering the code itself. (Obviously, the code must be considered in finding and correcting errors.) This is known as *black box* testing. Usually, a module is small enough to allow some additional testing based on logic paths in the code. Test cases might be designed that force the module through certain statements or sequences of statements. For example, the tester could make sure that both the *then* and the *else* clause of an *if* statement are executed. This is called *white box* testing.

Unit tests are often conducted by the programmer who writes each module. However, there are many advantages to having other people involved in the unit testing. From the programmer's point of view, a test is successful if it shows that the program works correctly. However, the purpose of testing is to reveal bugs in the software under test—thus, a test *should* be considered successful if it discloses a flaw. The programmer who writes a piece of code may be too close to it, and too psychologically invested in its success, to design rigorous tests.

After modules are tested individually, they must be tested in combination with each other to be sure that the interfaces are correct. This is known as *integration testing*. Except in relatively simple systems, the integration testing is usually done using an *incremental* approach. The system is built up by adding one module (or a small number of closely related modules) at a time, and the partial system is tested at each stage. Modules are usually added to the partial system using either a bottom-up or top-down ordering; these strategies are described in the following sections.

After all of the modules have been integrated into a complete system, the final phase, *system testing*, can begin. The goal of this testing is to verify that the entire system meets all of its specifications and requirements. System testing often takes place in two or more stages. *Alpha testing* is performed by the organization that developed the system, before it is released to any outside users. *Beta testing* or *field testing* involves placing the system into actual use in a limited number of environments. This sort of customer-performed testing often turns up problems that were missed by the system developers. Finally, customers may perform their own *acceptance testing* to decide whether or not to accept delivery of the system.

8.5.2 Bottom-Up Testing

The term *bottom-up testing* describes one common sequence in which the modules of a system undergo unit testing and are integrated into a partial system. In the case of a procedural system like the one shown in in Fig. 8.11(a), the term *bottom-up* refers to the hierarchical calling structure. The modules at the lowest level of the hierarchy (i.e., farthest from the root) are tested first, then the modules at the next higher level, and so on.

Thus, in the structure of Fig. 8.11(a), we might first perform unit testing on the modules Access_symtab and P2_search_optab. Then we might unit test module P2_assemble_inst. After this unit testing, we could combine these three modules into a partial system and perform integration testing on them. The other modules at level 3 of the hierarchy would also be unit tested individually. Then they would be integrated together to form Pass_1 and Pass_2, and these larger subsystems would be tested. Finally, the "driver" routine for the assembler would be unit tested, and all of the modules would be combined for system testing.

In the case of an object-oriented system, the situation may not be as clear. In general, bottom-up testing for such a system would begin with objects that are *passive*—that is, objects that do not invoke methods on other objects. In Fig. 8.14, for example, the objects Symbol_table, Opcode_table, Object_program, and Assembly_listing might be tested individually first. The sequence would then continue with other objects that invoke methods only on objects that have already been tested. Thus, in Fig. 8.14 the next object to be tested would be Source_line. Finally the object Source_program would be tested.

During the unit testing of individual modules and the integration testing of partial systems, it is necessary to simulate the presence of the remainder of the system. This can be done by writing a *test driver* program for each module. Figure 8.16 shows the outline of a simple test driver for the procedure Access_symtab from Fig. 8.11. This driver reads test cases (i.e., sets of calling parameters) that are supplied by the person performing the test and calls Access_symtab with these parameters. It then displays the results returned by the procedure, so that these can be compared with the correct input.

Essentially the same process would be used to test the behavior of an object. Each test case would include a specification of a method to be invoked on the object, and any parameters that are passed with the invocation. After invoking the method, the test driver would display any values that were returned from the invocation. The details of doing this depend on the syntax of the particular object-oriented language being used.

Bottom-up testing is the most frequently used strategy for unit testing and module integration. Test cases are delivered directly to the module, instead of being passed through the rest of the system. Thus, it is relatively easy to test a large variety of different conditions. Bottom-up testing also allows for the

```
program test_symtab (input, output);
var
   ...
begin
  while not eof(input) do
    begin
      readln (request_code, symbol, address);
                              { read test case from input }
      Access_symtab(request_code, return_code, symbol, address);
                              { call Access_symtab with test case }
      writeln(request_code, return_code, symbol, address);
                              { display result }
    end;
  ...
end.
```

Figure 8.16 Test driver for Access_symtab module.

simultaneous unit testing and integration of many different low-level modules in the system. This can be of real benefit in meeting project deadlines.

However, bottom-up testing has been criticized by a number of authors. The most frequent objection is that, with bottom-up testing, design errors that involve interfaces between modules are not discovered until the later stages of testing. When such errors are discovered, fixing them can be very expensive and time-consuming. In some complex systems, it can also be difficult to write drivers that exactly simulate the environment of the unit or partial system being tested.

8.5.3 Top-Down Testing

In *top-down testing* of a procedural system, modules are unit tested and integrated into partial systems beginning at the highest level of the hierarchical structure. For example, in the structure shown in Fig. 8.11(a) the first module to be unit tested would be the main routine for the assembler (the root node in the tree structure). Next, the main routines for Pass_1 and Pass_2 would be tested individually, and these three routines would be integrated together. The modules at the next level would be individually tested and integrated into the partial system being developed, and this process would continue until the system is complete.

In the case of an object-oriented system, top-down testing would begin with the object that is the primary focus of control for the system. For example, the interaction diagram in Fig. 8.15 indicates that the primary focus of control is the object Source_program. Testing would then continue with other objects whose methods are invoked by the primary object. In the case of Fig. 8.15, the next object to be tested would probably be Source_line. Finally, the remaining

objects in the system would be tested. Note that this is the reverse of the order produced by bottom-up testing.

During the top-down testing of modules and partial systems, we must simulate the presence of lower-level modules. This is done by writing *stubs* for the modules to be simulated. A stub contains only enough code to allow communication with the higher-level module being tested. For example, the stub for a module that computes a data value might return a constant value or a random value in some range, regardless of its input. A stub might also write a message that indicates that the module has been executed and that shows the parameters it was passed. In some cases, the stub need do nothing except exit back to the calling procedure. Figure 8.17 shows sample stubs for some of the modules in Fig. 8.11.

```
procedure P1_assign_loc (.....);
   var
      ...
   begin
      next_locctr := curr_locctr + 3;
   end;

procedure P2_search_optab (.....);
   var
      ...
   begin
      if mnemonic = 'LDA' then
         begin
           return_code := 0;        {mnemonic found in opcode table}
           opcode := '00';          {machine opcode = hex 00}
         end
      else
         begin
           return_code := 1;        {mnemonic not found in table}
           opcode := 'FF';          {set machine opcode to hex FF}
         end;
   end;

procedure P2_write_obj (.....);
   var
      ...
   begin
      writeln('*** P2_write_obj executed ***');
   end;
```

Figure 8.17 Sample stubs for modules of assembler.

The same process can be used to simulate the behavior of an object that has not yet been implemented. The object being simulated would contain a stub like those in Fig. 8.17 for each method it defines. The simulated object would include only enough code to allow communication with the object that invokes its methods. The details of doing this depend on the syntax of the particular object-oriented language being used.

Top-down testing detects design errors that involve module interfaces at an earlier stage than bottom-up testing. Thus, such errors can be fixed with less wasted time and effort. However, the testing process is more difficult to plan and execute. In some systems, it is very hard to devise input data that, when passed through the system structure, will adequately test all situations in a low-level module. In addition, top-down testing is very sensitive to delays in the planned schedule of tests. If one high-level module takes longer than anticipated to debug, the testing of many lower-level modules is also delayed.

In many cases, a combination of the bottom-up and top-down approaches is used in practice. For example, some of the most critical low-level modules could be tested first, and then the rest of the system could be integrated from the top down. Another approach uses both stubs and test drivers for the unit testing process; the resulting modules can be integrated together in any convenient sequence. Further discussions of bottom-up and top-down testing strategies can be found in Sommerville (1996).

EXERCISES

Section 8.2

1. Write a complete set of input specifications for a SIC (standard version) assembler, as described in Section 2.1. You may make any decisions about requirements that you feel are appropriate.
2. Write a complete set of specifications for an absolute loader, as described in Section 3.1.
3. List implementation restrictions that would be appropriate for a SIC/XE assembler that incorporates all of the features described in Sections 2.2 and 2.3.
4. What quality specifications might be appropriate for an operating system for a personal computer?
5. Write a decision table that specifies how bits n and i should be set in a SIC/XE Format 3 instruction (see Section 2.2).
6. Write a decision table specification for determining whether an expression specifies an absolute or relative value (see Section 2.3.3).

7. List input error conditions derived from the specifications that you wrote in Exercises 1 and 2.

Section 8.3

1. Draw a data flow diagram for an absolute loader (see Section 3.1).
2. Draw a data flow diagram for a linking loader (see Section 3.2).
3. Draw a data flow diagram for a linkage editor (see Section 3.4.1).
4. Draw a data flow diagram for a one-pass macro processor (see Section 4.1).
5. Draw a data flow diagram for a one-pass assembler that generates the object program in memory (see Section 2.4.1).
6. Modify the data flow diagram in Fig. 8.7(d) for a SIC/XE assembler that supports literals (see Sections 2.2 and 2.3.1).
7. Modify the data flow diagram in Fig. 8.7(d) for a SIC/XE assembler that supports program blocks (see Sections 2.2 and 2.3.4).
8. Write descriptions, at a suitable level of detail, for the data objects and processing actions in Fig. 8.7(d).
9. What information is "hidden" by module P1_read_source in Fig. 8.10?
10. Draw a data flow diagram for a Modularized Macroprocessor design.
11. Draw a data flow diagram for a Modularized Linking loader design.
12. Extend the Fig. 8.10 for program block and control section of assemblers.
13. Divide module P2_assemble_inst in Fig. 8.10 into a set of smaller modules, and specify the interfaces between these new modules.
14. Complete the interface description for P1_read_source that is given in Fig. 8.11 by specifying the representation, contents, and use of all of the parameters involved.
15. Complete the interface description for Access_symtab that is given in Fig. 8.11 by specifying the representation, contents, and use of all of the parameters involved.
16. Suppose that the execution of Pass_1 is "driven" by P1_read_source as described in the text. Specify the calling structure and parameters for this new organization, at the same level of detail as shown in Figs. 8.10 and 8.11.

17. Write a complete set of specifications for module P1_assign_sym.
18. Write a complete set of specifications for module Access_symtab.
19. Which of the parameters in Fig. 8.11 would you consider making global or common to two or more modules? Justify your choice.
20. Design a set of modules for a linking loader (see Section 3.2).
21. Design a set of modules for a one-pass macro processor (see Section 4.1).
22. Design a set of modules for a one-pass assembler that generates the object program in memory (see Section 2.4.1).

Section 8.4

1. Draw an object diagram and an interaction diagram for an absolute loader (see Section 3.1).
2. Draw an object diagram and an interaction diagram for a linking loader (see Section 3.2).
3. Draw an object diagram and an interaction diagram for a linkage editor (see Section 3.4.1).
4. Draw an object diagram and an interaction diagram for a one-pass macro processor (see Section 4.1).
5. Draw an object diagram and an interaction diagram for a one-pass assembler that generates the object program in memory (see Section 2.4.1).
6. Modify the object diagram in Fig. 8.14 for a SIC/XE assembler that supports literals (see Sections 2.2 and 2.3.1).
7. Modify the object diagram in Fig. 8.14 for a SIC/XE assembler that supports program blocks (see Sections 2.2 and 2.3.4).
8. List as many objects as you can that would be appropriate to consider in the design of a compiler. Briefly describe the contents of each object, and indicate what methods it would define.
9. List as many objects as you can that would be appropriate to consider in the design of an operating system. Briefly describe the contents of each object, and indicate what methods it would define.
10. Select a data structure for representing the instance variables of the object Source_line.

11. Select a data structure for representing the instance variables of the object Assembly_listing.
12. Write algorithms for implementing the methods of the object Source_line.
13. Write algorithms for implementing the methods of the object Assembly_listing.
14. How would the implementations of the Search method be different in the classes Hash_table and Binary_search_tree?
15. How would the implementations of the Enter method be different in the classes Hash_table and Binary_search_tree?
16. What methods might be implemented in the class Hash_table that would not be found in the class Binary_search_tree?
17. What methods might be implemented in the class Binary_search_tree that would not be found in the class Hash_table?
18. What other kinds of data structures might be defined as subclasses of the base class Searchable_data_structures [see Fig. 8.12(c)]. How would the implementations of these classes differ from each other? What methods (if any) would be unique to each derived class?

Section 8.5

1. Outline a test driver for module P1_read_source (see Figs. 8.10 and 8.11).
2. Devise a set of test cases to be used in the unit testing of P1_read_source with the test driver you outlined in Exercise 1.
3. Outline a test driver for module P2_write_obj (see Figs. 8.10 and 8.11).
4. Devise a set of test cases to be used in the unit testing of P2_write_obj with the test driver you outlined in Exercise 3.
5. Outline a test driver for the object named Object_program (see Figs. 8.14 and 8.15).
6. Devise a set of test cases to be used in the unit testing of Object_program with the test driver you outlined in Exercise 5.
7. Outline a test driver for the object named Source_line (see Figs. 8.14 and 8.15).

8. Devise a set of test cases to be used in the unit testing of Source_line with the test driver you outlined in Exercise 7.
9. Write a stub for Access_symtab to be used in the top-down testing of the modular structure in Figs. 8.10 and 8.11.
10. Write a stub for P1_read_source to be used in the top-down testing of the modular structure in Figs. 8.10 and 8.11.
11. Write stubs for the methods of the object Opcode_table (see Figs. 8.14 and 8.15).
12. Write stubs for the methods of the object Source_line (see Figs. 8.14 and 8.15).