```
Loop: if not C then goto End;
      S;
      goto Loop;
End:
```

Fragment H1. Typical translation of "`while C do S`".

```
        goto Test;
Loop:   S;
Test:   if C then goto Loop;
End:
```

Fragment H2. Efficient translation of "`while C do S`".

```
Sum := 0;
for I := 1 to N do
  Sum := Sum + X[I];
Greater := Sum > CutOff
```

Fragment J1. Sum first then compare.

```
I := 1;
Sum := 0;
while I <= N and Sum <= CutOff do
  begin
  Sum := Sum + X[I];
  I := I+1
  end;
Greater := Sum > CutOff
```

Fragment J2. Compare as we sum.

```
function CharType(X: char): integer;
  begin
  if X = ' ' then return 1
    else if   ('A' <= X and X <= 'I')
           or ('J' <= X and X <= 'R')
           or ('S' <= X and X <= 'Z') then
         return 2
    else if '0' <= X and X <= '9' then return 3
    else if   X = '+' or X = '/' or X = '-'
           or X = ',' or X = '('
           or X = ')' or X = '=' then return 4
    else if X = '*' then return 5
    else if X = '"' then return 6
    else return 7
  end;
```

Fragment K1. A character recognizer.

```
function CharType(X: char): integer;
  begin
  if X = ' ' then return 1
    else if X = '*' then return 5
    else if X = '"' then return 6
    else if '0' <= X and X <= '9' then return 3
    else if   ('A' <= X and X <= 'I')
           or ('J' <= X and X <= 'R')
           or ('S' <= X and X <= 'Z') then
         return 2
    else if   X = '+' or X = '/' or X = '-'
           or X = ',' or X = '('
           or X = ')' or X = '=' then return 4
    else return 7
    end;
```

Fragment K2. Order of tests changed.

```
function CharType(X: char): integer;
  begin
  if X = ' ' then return 1
    else if X = '*' then return 5
    else if X = '"' then return 6
    else if '0' <= X and X <= '9' then return 3
    else if    ('A' <= X and X <= 'I')
           or ('J' <= X and X <= 'R')
           or ('S' <= X and X <= 'Z') then
         return 2
    else if   X = '+' or X = '/' or X = '-'
           or X = ',' or X = '('
           or X = ')' or X = '=' then return 4
    else return 7
    end;
```

Fragment K2. Order of tests changed.

```
function CharType(X: char): integer;
  begin
  return TypeTable[ord(X)]
  end;
```

Fragment K3. Character recognition by table lookup.

```
V := LogicalExp;
S1;
if V then
    S2
  else
    S3
```

Fragment L1. Code with boolean variable V.

```
if LogicalExp then
    begin
    S1;
    S2
    end
  else
    begin
    S1;
    S3
    end
```

Fragment L2. Boolean variable V removed.

```
procedure ApproxTSTour;
  var
    I: PtPtr;
    UnVis: array [PtPtr] of PtPtr;
    ThisPt, HighPt, ClosePt, J: PtPtr;
    CloseDist, ThisDist: real;
  procedure SwapUnVis(I, J: PtPtr);
    var Temp: PtPtr;
    begin
    Temp := UnVis[I];
    UnVis[I] := UnVis[J];
    UnVis[J] := Temp
    end;

  begin
  (* Initialize unvisited points *)
  for I := 1 to NumPts do
  UnVis[I] := I;
  (* Choose NumPts as starting point *)
  ThisPt := UnVis[NumPts];
  HighPt := NumPts-1;
  (* Main loop of nearest neighbor tour *)
  while HighPt > 0 do
    begin
    (* Find nearest unvisited point to ThisPt *)
    CloseDist := maxreal;
    for I := 1 to HighPt do
      begin
      ThisDist := DistSqrd(UnVis[I],ThisPt);
      if ThisDist < CloseDist then
        begin
        ClosePt := I;
        CloseDist := ThisDist
        end
      end;
    (* Report this point *)
    ThisPt := UnVis[ClosePt];
    SwapUnVis(ClosePt,HighPt);
    HighPt := HighPt-1
    end
  end;
```

Fragment A4. Convert boolean array to pointer array.

```
(* Find nearest unvisited to ThisPt *)
ThisX := PtArr[ThisPt].X;
ThisY := PtArr[ThisPt].Y;
CloseDist := maxreal;
for I := 1 to HighPt do
  begin
  ThisDist := sqr(PtArr[UnVis[I]].X-ThisX)
              + sqr(PtArr[UnVis[I]].Y-ThisY);
  if ThisDist < CloseDist then
    begin
    ClosePt := I;
    CloseDist := ThisDist
    end
  end;
```

Fragment A5. Rewrite procedure in line and remove invariants.

```
function CharType(X: char): integer;
  begin
  return TypeTable[ord(X)]
  end;
```

Fragment K3. Character recognition by table lookup.

```
procedure ApproxTSTour;
  var I, J: PtPtr;
    Visited: array [PtPtr] of boolean;
    ThisPt, ClosePt: PtPtr;
    CloseDist: real;

  begin
  (* Initialize unvisited points *)
  for I := 1 to NumPts do
    Visited[I] := false;

  (* Choose NumPts as starting point *)
  ThisPt := NumPts;
  Visited[NumPts] := true;
  writeln('First city is ', NumPts);

  (* Main loop of nearest neighbor heuristic *)
  for I := 2 to NumPts do
    begin
  (* Find nearest unvisited point to ThisPt *)
    CloseDist := maxreal;
    for J := 1 to NumPts do
      if not Visited[J] then
        if Dist(ThisPt, J) < CloseDist then
          begin
          CloseDist := Dist(ThisPt, J);
          ClosePt := J
          end;
    (* Report closest point *)
    writeln('Move from', ThisPt, 'to', ClosePt);
    Visited[ClosePt] := true;
    ThisPt := ClosePt
    end;

  (* Finish tour by returning to start *)
  writeln('Move from', ThisPt, 'to', NumPts)
  end;
```

Fragment A1. Original code.

```
begin
ThisDist := Dist(ThisPt, J);
if ThisDist < CloseDist then
  begin
  CloseDist := ThisDist;
  ClosePt := J
  end
end
```

Fragment A2. Store ThisDist.

```
procedure A(..)                    procedureB(..)
  L0: coroutine B                      . . .
  L1: . . .                             M1: . . .
    resume B(..)                      resume A(..)
  L2: . . .                             M2: . . .
    resume B(..)                      resume A(..)
  L3: . . .                             M3: . . .
end A                              end B
```

Table 7-11: Two Coroutines Which Execute the Labeled Sequence: A starts, L0, B starts, M1, L1, M2, L2, M3, B ends, L3, A ends.

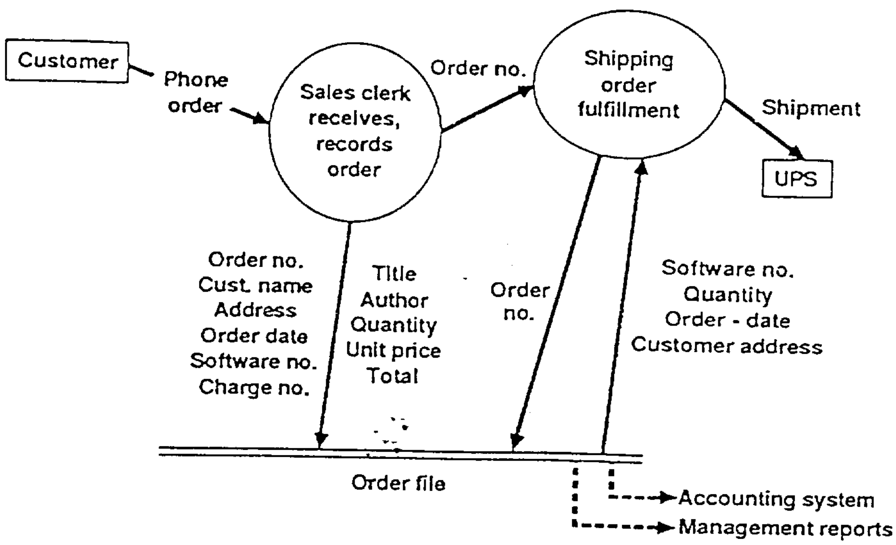

FIGURE 5.14
The Software Store—an example.

Steps 1 and 2: List data elements/group for the *Software Shop* example

(a)	(b)
Order no. Order date Customer name Address Charge no. Order list * Software no. Title Author Quantity Unit price Total price	ORDER - DATA Order no. Order date Customer name Address Charge no. ORDER - LIST Order no. Software no. Title Author Quantity Unit price Total price

* repeated 1 to *n* times

(a) Unnormalized structure.

(b) Two normalized structures.

FIGURE 5.32
Normalization.

ORDER-LIST (order-no, software-no, title, author, quantity, unit-price, total-price)

ORDER-DATA (order-no, order-date, customer-name, address, charge-no)

ORDER-DATA (order-no, order-date, customer-name, address, charge-no)
ORDER-LIST (order-no, software-no, quantity, total-price)
SOFTWARE-INFO (software-no, title, author, unit-price)

ORDER-DATA (order-no, order-date, customer-name, address, charge-no)
ORDER-LIST (order-no, software-no, quantity)
SOFTWARE-INFO (software-no, title, author, unit-price)

T1 List all software information for each order.
T2 Get customer name, address, etc., given order no.
T3 List price information given software no.
T4 Compute total sales for a particular date.
T5 Compare total sales ($) on one date with the same information from another date.

Characteristic	Transactions
Type	On-line, batch, software originated, human source
Frequency	Number of transactions/month
Use	Read, modify, add, delete
Data element list	Relationship to each transaction

FIGURE 5.33
The transaction matrix.

Transaction	T_1	T_2	T_3	T_4	T_5
Type	Batch	On-line	On-line	Batch	On-line
Frequency	200	600	1000	10	5
Use	R	R	R	R	R
Order no.	RK	RK		RK	RK
Order date	R			RK	RK
Customer name	R	R			
Address	R	R			
Charge no.	R				
Software no.	RK		RK		
Quantity	R			R	R
Title	R				
Author	R				
Unit price	R		R	R	R

R = read; K= key data element

FIGURE 5.34
Transaction matrix for *The Software Store* data base.

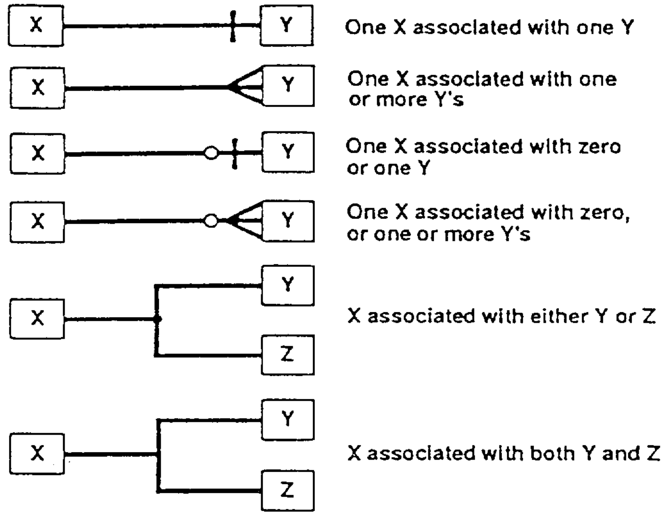

FIGURE 5.35
Entity relationship notation.

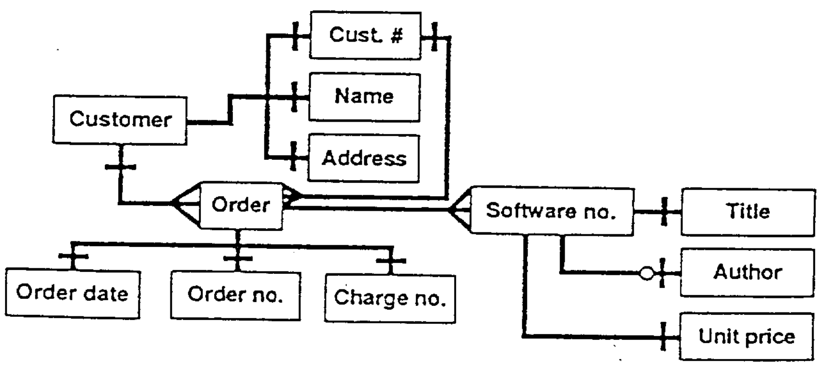

FIGURE 5.36
Entity relationship diagram for *The Software Store*.

The rules for forming regular expressions are quite simple:

1. Atoms: The basic symbols in the alphabet of interest form regular expressions.
2. Alternation: If R1 and R2 are regular expressions, then (R1 | R2) is a regular expression.
3. Composition: If R1 and R2 are regular expressions, then (R1 R2) is a regular expression.
4. Closure: If R1 is a regular expression, then (R1)* is a regular expression.
5. Completeness: Nothing else is a regular expression.

An alphabet of basic symbols provides the atoms. The alphabet is made up of whatever symbols are of interest in the particular application. Alternation, (R1 | R2), denotes the union of the languages (sets of symbol strings) specified by R1 and R2, and composition, (R1 R2), denotes the language formed by concatenating strings from R2 onto strings from R1. Closure, (R1)*, denotes the language formed by concatenating zero or more strings from R1 with zero or more strings from R1.

Observe that rules 2, 3, and 4 are recursive; i.e., they define regular expressions in terms of regular expressions. Rule 1 is the basis rule, and rule 5 completes the definition of regular expressions. Examples of regular expressions follow:

1. Given atoms a and b, then a denotes the set { a } and b denotes the set { b }.
2. Given atoms a and b, then (a | b) denotes the set { a, b }.
3. Given atoms a, b, and c, then ((a | b) | c) denotes the set { {a,b}, c}.
4. Given atoms a and b, then (a b) denotes the set { ab } containing one element ab.
5. Given atoms a, b, and c, then ((a b) c) denotes the set { abc } containing one element abc.
6. Given atom a, then (a)* denotes the set {e, a, aa, aaa, ...}, where e denotes the empty string.

Complex regular expressions can be formed by repeated application of recursive rules 2, 3, and 4:

1. (a (b | c)) denotes {ab, ac}.
2. (a | b)* denotes {e, a, b, aa, bb, ab, ba, aab, ...}.
3. ((a (b | c)))* denotes {e, ab, ac, abab, acac, abac, acab, ababac, ...}

Closure, (R1)*, denotes zero or more concatenations of elements from R1. A commonly used notation is (R1) + , which denotes one or more concatenations of elements in R1. The "*" and " + " notations are called the Kleene star and Kleene plus notations. They are named for their inventor.

Regular expressions can be given many different interpretations, and are thus useful in many different situations. For instance, ((a (b|c))) + might denote any of the following:

1. A data stream. If a, b, and c are input data symbols, then valid data streams must always start with an "a", followed by "b"s and "c"s in any order, but always interleaved by "a" and terminated by "b" or "c".
2. Message transmission. a, b, and c can be interpreted as message types such as resource request or release, job initiation request, or end of file. The regular expression then specifies legal sequences of messages.
3. Operation sequence. If a, b, and c represent procedures, then legal calling sequences are "a" followed by a call to "b" or "c" followed by zero or more returns to "a" followed by calls to "b" or "c". (The ambiguity of the preceding sentence illustrates the desirability of using formal notations.)
4. Resource flow. Symbols a, b, and c might denote system components such as a process or a user. The regular expression ((a (b|c))) + is associated with a resource such as a processor, a tape unit, a file, or a system table. The regular expression then states the a must get the resource first, then either b or c may have it after a releases it, and that a must always have the resource between allocations to b or c.

NEW creates a new stack.
PUSH adds a new item to the top of a stack.
TOP returns a copy of the top item.
POP removes the top item.
EMPTY tests for an empty stack.

Operation NEW yields a newly created stack. PUSH requires two arguments, a

```
SYNTAX:
  OPERATION   DOMAIN                 RANGE
  NEW         (  ) —>                STACK
  PUSH        (STACK,ITEM) —>        STACK
  POP         (STACK) —>             STACK
  TOP         (STACK) —>             ITEM
  EMPTY       (STACK) —>             BOOLEAN
AXIOMS:
(stk is of type STACK, itm is of type ITEM)
  (1) EMPTY(NEW) = true
  (2) EMPTY(PUSH(stk,itm)) = false
  (3) POP(NEW) = error
  (4) TOP(NEW) = error
  (5) POP(PUSH(stk,itm)) =  stk
  (6) TOP(PUSH(stk,itm)) =  itm
```

Figure 4.3 Algebraic specification of the LIFO property.

1. A new stack is empty.
2. A stack is not empty immediately after pushing an item onto it.
3. Attempting to pop a new stack results in an error.
4. There is no top item on a new stack.
5. Pushing an item onto a stack and immediately popping it off leaves the stack unchanged.
6. Pushing an item onto a stack and immediately requesting the top item returns the item just pushed onto the stack.

Table 4.8 A simple transition table

Current state	Current input	
	a	b
S0	S0	S1
S1	S1	S0
	Next state	

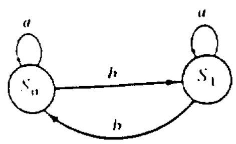

Figure 4.7 Transition diagram corresponding to Table 4.8.

Table 4.7 A two-dimensional event table

Mode	Event				
	E13	E37	E45	. . .	. . .
Start-up	A16	—	A14; A32		
Steady	X	A6, A2	—		
Shut-down	. . .	. . .	. . .		
Alarm	. . .	. . .	. . .		

Conditions	Decision Rules							
Condition 1	Y	Y	Y	Y	N	N	N	N
Condition 2	Y	Y	N	N	Y	Y	N	N
Condition 3	Y	N	Y	N	Y	N	Y	N
Actions	Action Entries							
Action 1	X						X	
Action 2		X				X		
Action 3			X		X			
Action 4				X				X

Generic Decision Table

2a

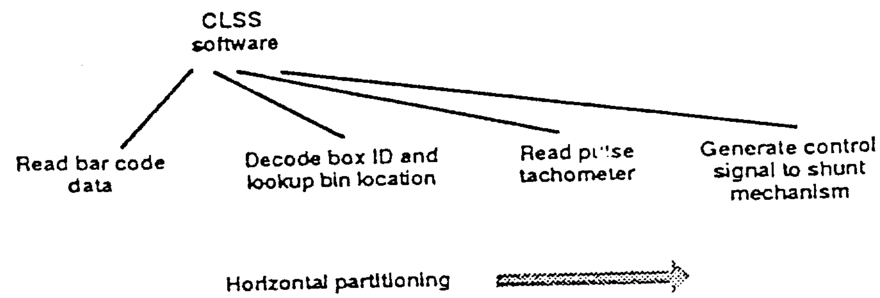

FIGURE 4.5
Horizontal partitioning.

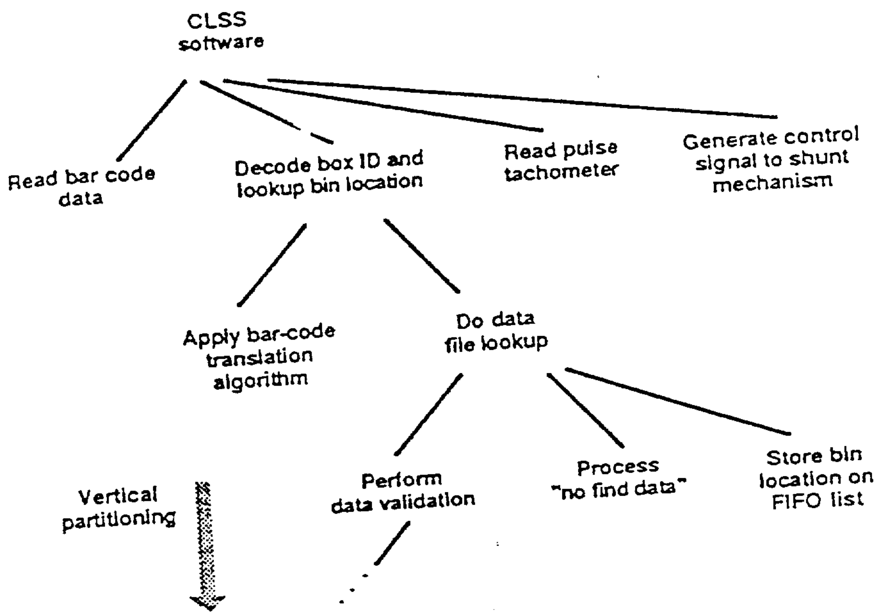

FIGURE 4.6.
Vertical partitioning.

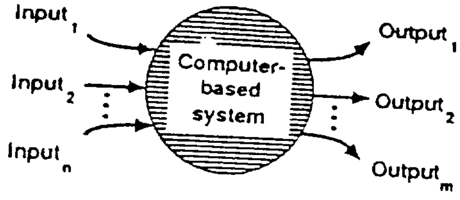

FIGURE 5.1
Information flow.

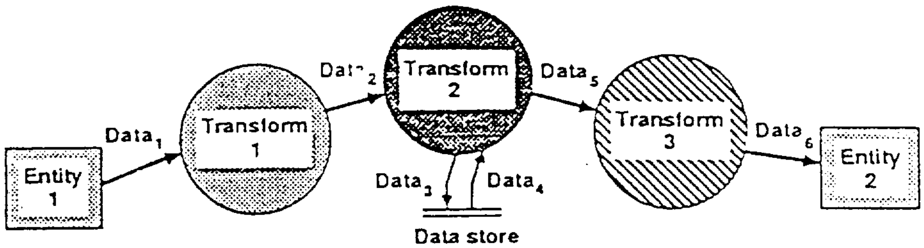

FIGURE 5.2
A data flow diagram (DFD).

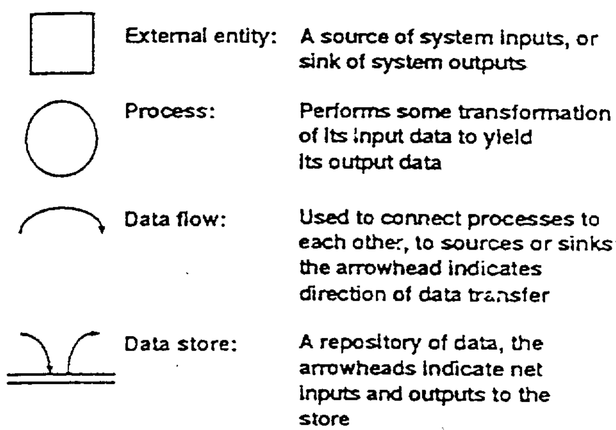

FIGURE 5.3
DFD symbology.

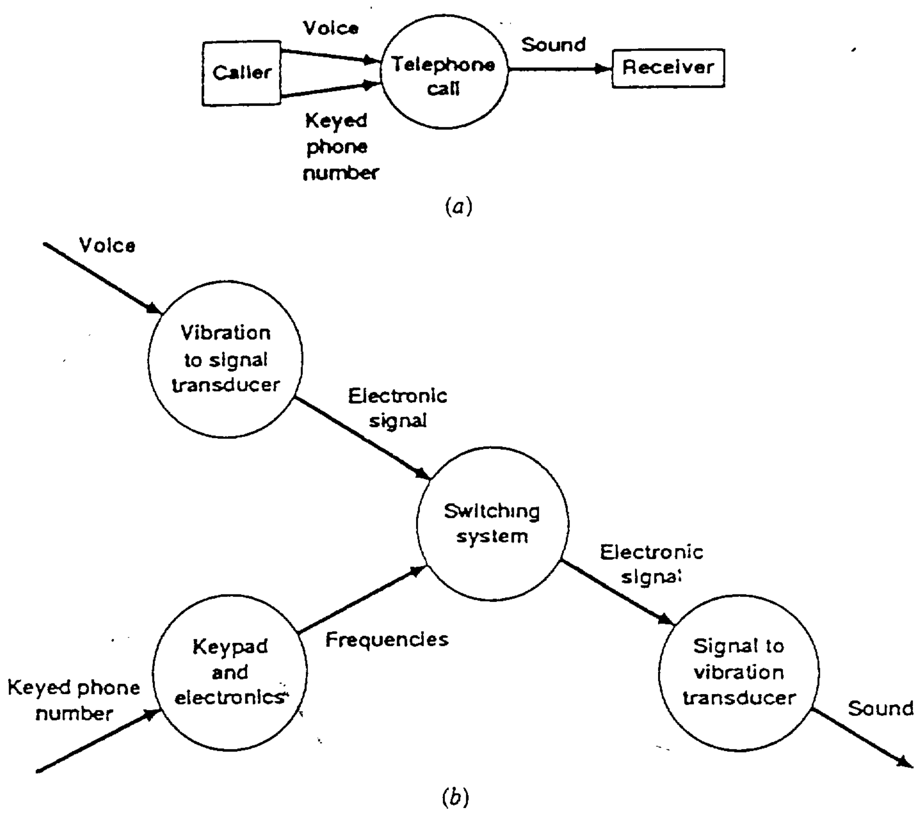

FIGURE 5.4
Example—a telephone call DFD. (*a*) Level 01 data flow diagram. (*b*) Level 02 data flow diagram.

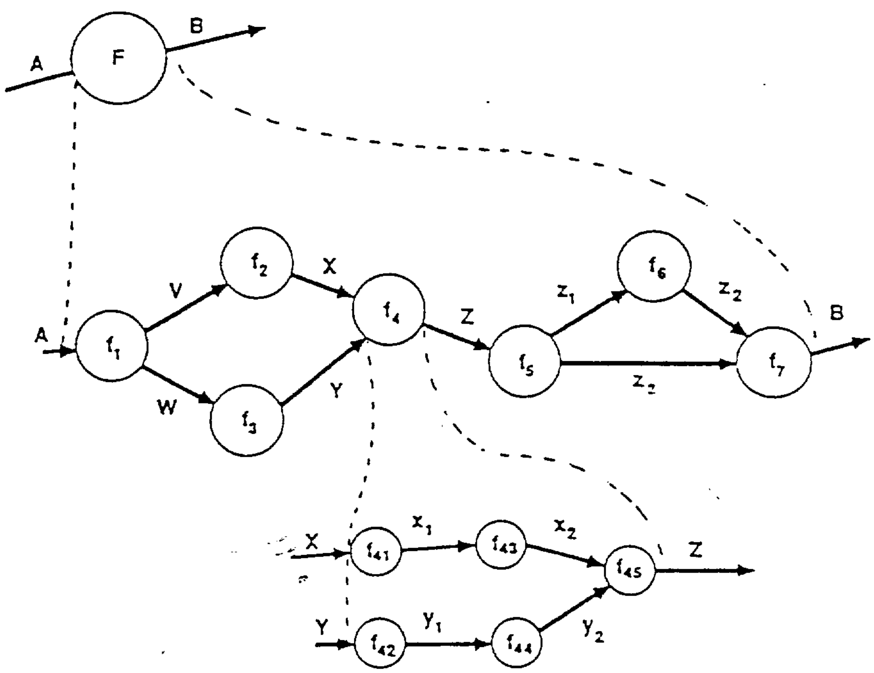

FIGURE 5.5
Information flow refinements.

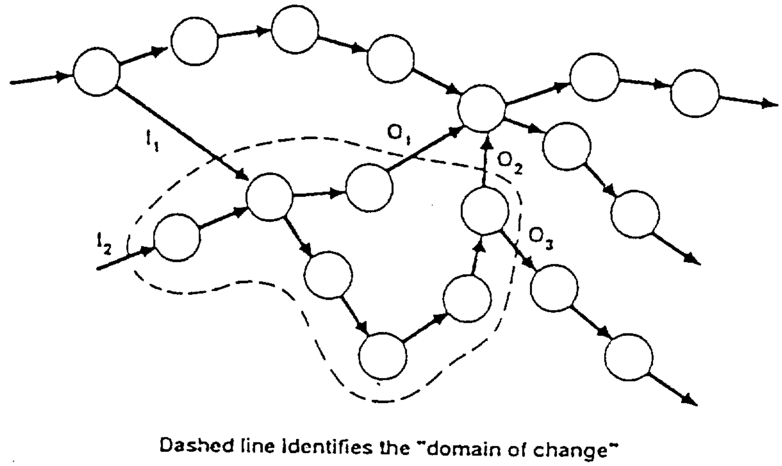

FIGURE 5.9
Data flow diagrams for existing systems.

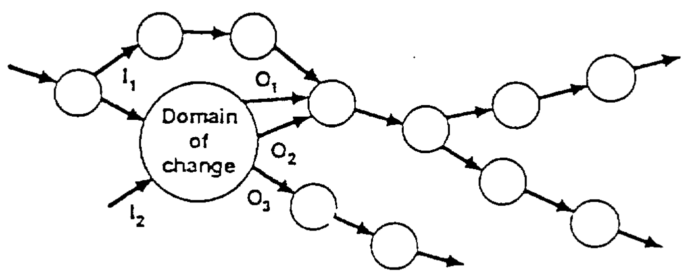

FIGURE 5.10
Remodel with domain of change isolated.

Data construct	Notation	Meaning
	=	Is composed of
Sequence	+	And
Selection	[|]	Either - or
Repetition	$\{\ \}^n$	*n* repetitions of
	( )	Optional data

FIGURE 5.11
Data dictionary notation.

keyed phone number = [local extension | outside number | 0]
local extension = [2001 | 2002 | ... | 2999 | conference set]
outside number = 9 + [local number | long distance no.]
local number = prefix + access number
long distance no. = (0) + area code + local number
conference set = $\{\# + \text{local extension} + \#(\#)\}_2^6$

Basic Data Dictionary Notation

SYMBOL	DEFINITION	EXAMPLE
=	IS EQUIVALENT TO IS COMPOSED OF	CUST-NAME = SURNAME AND FAMILY-NAME CUST-NAME = SURNAME + FAMILY-NAME
+	AND	ADDRESSEE = CUST-NAME + ADDRESS
[ ]	EITHER-OR*	ADDRESS = [PO BOX NO / STREET ADDRESS] + STATE + ZIP
{ }	ITERATIONS OF†	PLAYER-ROSTER = {PLAYER-NAME + PLAYER-NO}
( )	OPTIONAL	PLAYER-NAME = SURNAME + (MIDDLE-INITIAL) + FAMILY-NAME

*You may use a vertical bar between items to express the definition on a single line (see Fig. 5.6).

†Default limits are 0 and ∞; that is, { X } means there may be as few as zero or an undefined number of X's. Use " " to denote literals, and * * to denote comments.

```
MAILING-LIST   = {CUSTOMER-NAME + MAILING-ADDRESS}

CUSTOMER-NAME  = (TITLE) + GIVEN-NAME + FAMILY-NAME

TITLE          = ["MR."|"MS."|"MRS."|"RESIDENT"]
                 * THE USE OF "RESIDENT" AS A TITLE
                 WILL BE DELETED AS OF DEC. 1, 1980 *

GIVEN-NAME     = 1 {ALPHABETIC-CHARACTER} 16

FAMILY-NAME    = 1 {ALPHABETIC-CHARACTER} 32
```

Examples of data dictionary notation in linear format.

Table 4.2 A Data dictionary entry

NAME:	Create
WHERE USED:	SDLP
PURPOSE:	Create passes a user-created design entity to the SLP processor for verification of syntax.
DERIVED FROM:	User Interface Processor
SUBITEMS:	Name Uses Procedures References
NOTES:	Create contains one complete user-created design entity.