3.2 System Process Requirements

Introduction

In this topic, our focus will be on one **tool** that is used to coherently represent the information gathered as part of **requirements determination—data flow diagrams.** Data flow diagrams enable you to model how data flow through an **information system, the relationships among the data flows, and how data come to be stored at specific locations**. Data flow diagrams also show the processes that change or transform data. Because data flow diagrams concentrate on the movement of data between processes, these diagrams are called **process models.**

Decision tables allow you to represent the conditional logic that is part of some data flow diagram processes.

Process Modeling

Process modeling involves **graphically** representing the **functions**, or **processes**, that **capture**, **manipulate**, **store**, and **distribute** data between a system and its environment and between components within a system. A common form of a process model is a **data flow diagram (DFD). DFDs**, the traditional process modeling technique of structured analysis and design and one of the techniques most frequently used today for process modeling.

Modeling a system's Process for structured Analysis:

As Figure 7-1shows, the analysis phase of the systems development life cycle has two **subphases** :

requirements determination and

requirements structuring.

The analysis team enters the requirements structuring phase with an abundance of information gathered during the requirements determination phase.

During requirements structuring, you and the other team members must organize the information into a meaningful representation of the information system that currently exists and of the requirements desired in a replacement system.

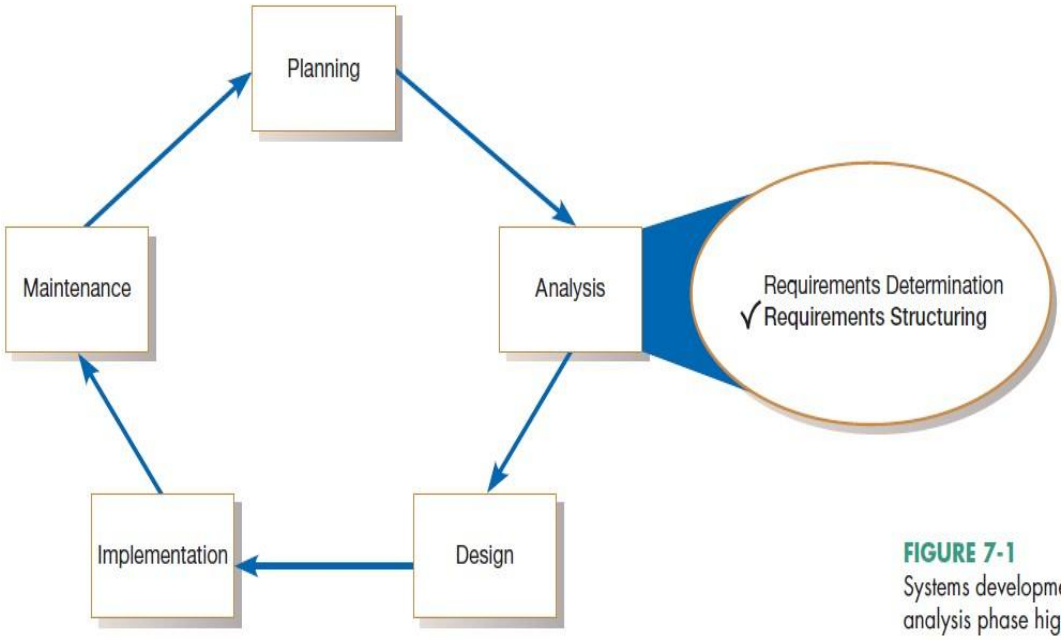


Figure 7-1 Systems development life cycle with the analysis phase highlighted

Deliverables and Outcomes:

In structured analysis, the primary deliverables from process modeling are a set of **coherent**, **interrelated DFDs**. Table 7-1 provides a more detailed list of the deliverables that result when DFDs are used to study and document a system's processes.

First, a **context diagram** shows the **scope of the system**, indicating which elements are inside and which are outside the system.

Second, DFDs of the system **specify which processes** move and transform data, accepting inputs and producing outputs.

These diagrams are developed with sufficient detail to understand the current system and to eventually determine how to convert the current system into its replacement. **Finally**, entries for all of the objects included in all of the diagrams are included in the **project dictionary** or **CASE repository**.

TABLE 7-1 Deliverables for Process Modeling

1. Context DFD
2. DFDs of the system (adequately decomposed)
3. Thorough descriptions of each DFD component

Data Flow Diagramming Mechanics:

DFDs are versatile diagramming tools. With only **four symbols**, you can use DFDs to represent both **physical** and **logical** information systems. DFDs are not as good as flowcharts for depicting (to represent or show something in a picture or story) the details of physical systems.

There are two different standard sets of DFD symbols (see Figure 7-2); each set consists of four symbols that represent the same things: **data flows**, **data stores**, **processes**, and **sources/sinks** (or external entities).

A **data store** is data at rest. A data store may represent one of many different physical locations for data; for example, a file folder, one or more computer-based file(s), or a notebook. A **data store** might contain

data about **customers**, **students**, **customer orders**, or **supplier invoices**.

A **process** **is** the work or actions performed on data so that they are transformed, stored, or distributed. When modeling the data processing of a system, it does not matter whether a process is performed manually or by a computer.

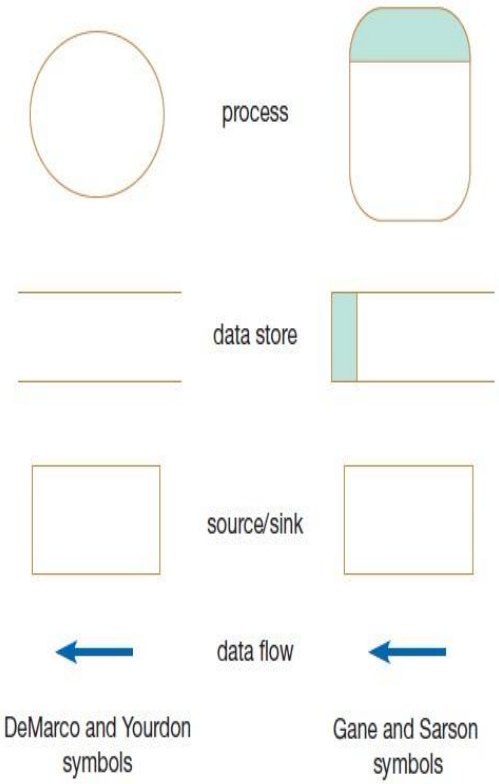


FIGURE 7-2
Comparison of DeMarco and Yourdon and Gane and Sarson DFD symbol sets

Finally, a **source/sink** **is the origin** and/or destination of the data. Sources/sinks are sometimes referred to as external entities because they are outside the system. Once processed, data or information leave the system and go to some other place. Sources and sinks are outside the system we are studying.

The symbols for each set of DFD conventions are presented in Figure 7-2. In both conventions, a data flow is represented as an **arrow**. The arrow is labeled with a meaningful name for the data in motion; for example, **Customer Order**, **Sales Receipt**, or **Paycheck**. The name represents the aggregation of all the individual elements of data moving as part of one packet, that is, all the data moving together at the same time.

Sources/Sinks are always outside the information system and define the boundaries of the system. Data must originate outside a system from one or more sources, and the system must produce information to one or more sinks (these are principles of open systems, and almost every information system is an open system). If any data processing takes place inside the source/sink, it is of no interest because this processing takes place outside the system we are diagramming. A source/sink might consist of the following:

Developing DFD

The information system is represented as a DFD in Figure 7-4. The highest-level view of this system, shown in the figure, is called a **context diagram. You will notice** that this context diagram contains only one process, **no data stores**, four data flows, and **three sources/sinks**. The single process, labeled 0, represents the entire system; all **context diagrams** have only **one process, labeled 0**. The sources/sinks represent the environmental boundaries of the system. Because **the data stores of the system are conceptually inside one process**, data stores **do not appear on a context diagram.** As you can see in Figure 7-5, we have identified four separate processes. The main processes represent the major

functions of the system, and these major functions correspond to actions such as the following:

1. Capturing data from different sources (e.g., **Process 1.0**)
2. Maintaining data stores (e.g., **Processes 2.0 and 3.0**)
3. Producing and distributing data to different sinks (e.g., **Process 4.0**)
4. High-level descriptions of data transformation operations (e.g., **Process1.0**)

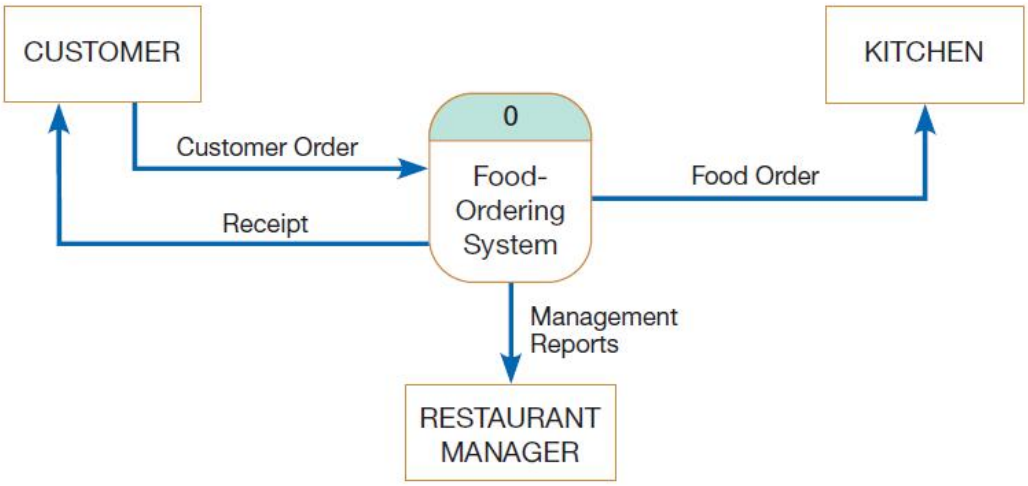


FIGURE 7-4
Context diagram of Hoosier Burger's food-ordering system

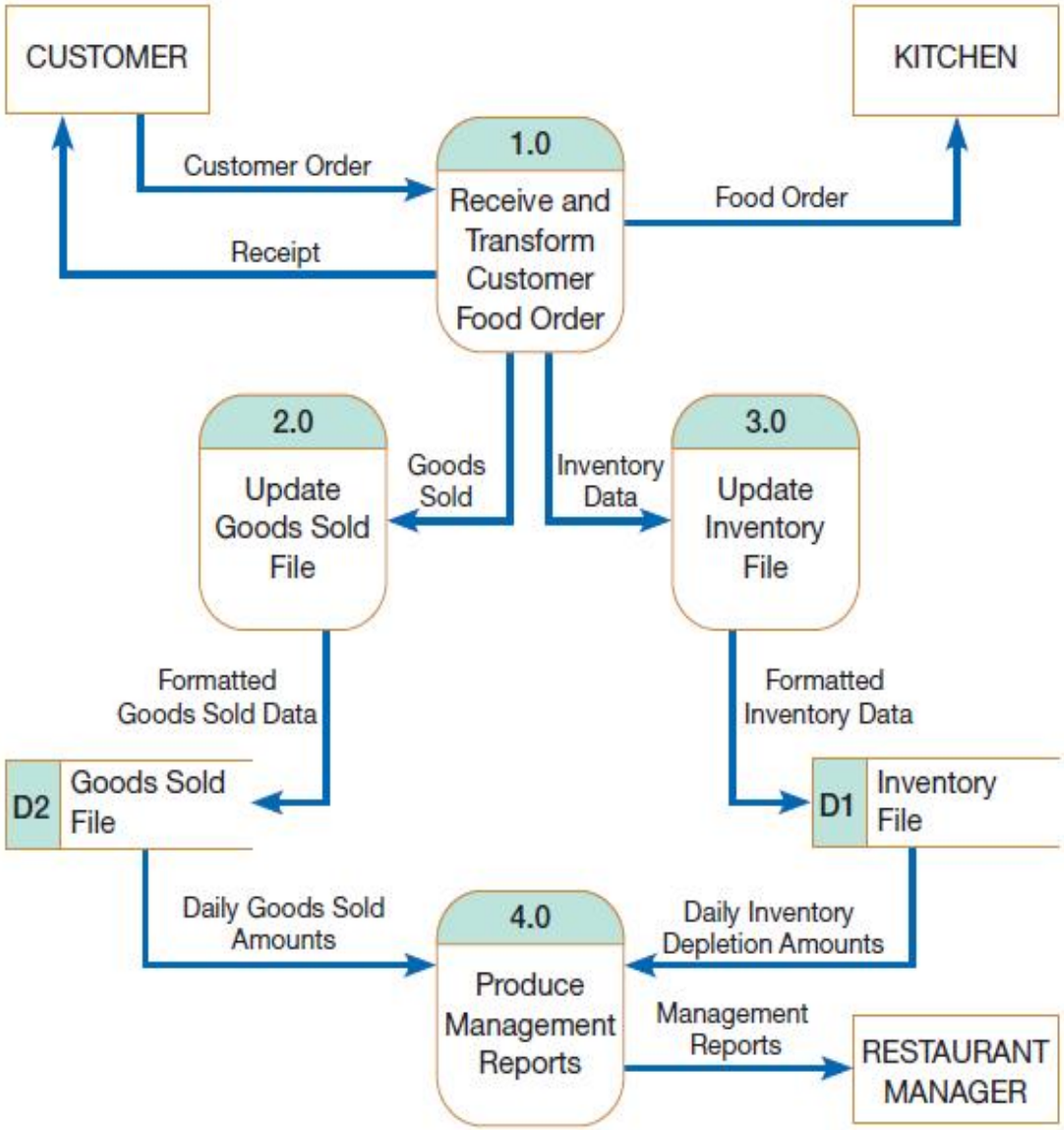


FIGURE 7-5
Level-0 DFD of Hoosier Burger's food-ordering system

We see that the **system begins** with an order from a customer, as was the case with the context diagram. In the first process, labeled 1.0, we see that the customer order is processed. The result is four streams, or flows, of data:

- the food order is transmitted to the kitchen,
- the customer order is transformed into a list of goods sold,
- the customer order is transformed into inventory data, and
- the process generates a receipt for the customer.

Notice that the sources/sinks are the same in the context diagram and in this diagram: the customer, the kitchen, and the restaurant's manager.

This diagram is called a **level-0 diagram because it represents the primary individual processes in the** system at the highest possible level. Each process has a number that ends in .0 (corresponding to the level number of the DFD).

Two of the data flows generated by the first process, Receive and Transform Customer Food Order, go to external entities, so we no longer have to worry about them. We are not concerned about what happens outside our system. Let's trace the flow of the data represented in the other two data flows. **First**, the **data labeled Goods Sold go to Process 2.0, Update Goods Sold File. The output for this process is labeled Formatted Goods Sold Data.** This output updates a data store labeled Goods Sold File. If the customer order was for two cheeseburgers, one order of fries, and a large soft drink, each of these categories of goods sold in the data store would be incremented appropriately. **The Daily Goods Sold Amounts are then used as input to Process 4.0, Produce Management Reports**. Similarly, the remaining data flow generated by Process 1.0, Inventory Data, serves as input for Process 3.0, Update Inventory File. This process updates the Inventory File data store, based on the inventory that would have been used to create the customer order. For example, an order of two cheeseburgers would mean that Hoosier Burger now has two fewer hamburger patties, two fewer burger buns, and four fewer slices of American cheese.

The Daily Inventory Depletion Amounts are then used as input to Process 4.0. The data flow leaving Process 4.0, Management Reports, goes to thesink Restaurant Manager. Figure 7-5 illustrates several important concepts about information movement. **Consider the data flow Inventory Data moving from Process 1.0 to Process 3.0**. We know from this diagram that Process 1.0 producesthis data flow and that Process 3.0 receives it. However, **we do not know the timing of when this data flow is produced**, how **frequently it is produced**, or what **volume of data is sent**. Thus, this **DFD hides many physical characteristics of the system it describes**.

We do know, however, that this data flow is needed by Process 3.0 and that Process 1.0 provides these needed data.

Also implied by the Inventory Data data flow is that whenever Process1.0 produces this flow, Process 3.0 must be ready to accept it. Thus, Processes 1.0 and 3.0 are coupled with each other. In contrast, consider the link between Process 2.0 and Process 4.0. The output from Process 2.0, Formatted Goods Sold Data, is placed in a data store and, later, when Process 4.0 needs such data, it reads Daily Goods Sold Amounts from this data store. In this case, Processes 2.0 and 4.0 are decoupled by placing a buffer, a data store, between them. Now, each of these processes can work at their own place, and Process 4.0 does not have to be ready to accept input at any time. Further, the Goods Sold File becomes a data resource that other processes could potentially draw upon for data.

Data Flow Diagramming Rules

TABLE 7-2 Rules Governing Data Flow Diagramming

Process:

A. No process can have only outputs. It would be making data from nothing (a miracle). If an object has only outputs, then it must be a source.

B. No process can have only inputs (a black hole). If an object has only inputs, then it must be a sink.

C. A process has a verb phrase label.

Data Store:

D. Data cannot move directly from one data store to another data store. Data must be moved by a process.

E. Data cannot move directly from an outside source to a data store. Data must be moved by a process that receives data from the source and places the data into the data store.

F. Data cannot move directly to an outside sink from a data store. Data must be moved by a process.

G. A data store has a noun phrase label.

Source/Sink:

H. Data cannot move directly from a source to a sink. It must be moved by a process if the data are of any concern to our system. Otherwise, the data flow is not shown on the DFD.

I. A source/sink has a noun phrase label.

Data Flow:

J. A data flow has only one direction of flow between symbols. It may flow in both directions between a process and a data store to show a read before an update. The latter is usually indicated, however, by two separate arrows because these happen at different times.

K. A fork in a data flow means that exactly the same data goes from a common location to two or more different processes, data stores, or sources/sinks (this usually indicates different copies of the same data going to different locations).

L. A join in a data flow means that exactly the same data come from any of two or more different processes, data stores, or sources/sinks to a common location.

M. A data flow cannot go directly back to the same process it leaves. There must be at least one other process that handles the data flow, produces some other data flow, and returns the original data flow to the beginning process.

N. A data flow to a data store means update (delete or change).

O. A data flow from a data store means retrieve or use.

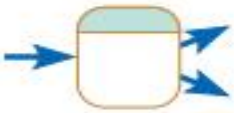
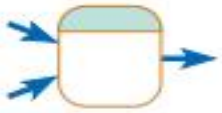
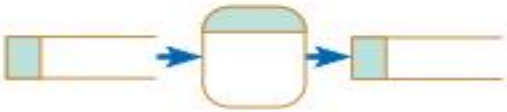
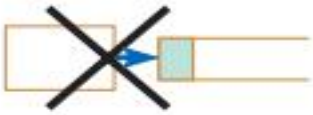
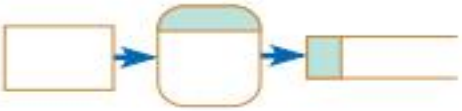
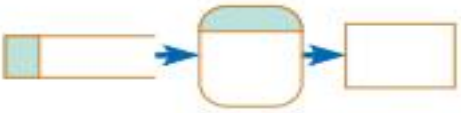
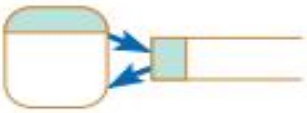
P. A data flow has a noun phrase label. More than one data flow noun phrase can appear on a single arrow as long as all of the flows on the same arrow move together as one package.

(*Source:* Based on Celko, 1987.)

Decomposition of DFDs

In the earlier example of Hoosier Burger's food-ordering system, we started with a high-level context diagram. Upon thinking more about the system, we saw that the larger system consisted of four processes. The act of going from a single system to **four component processes** is called (functional) decomposition. **Functional decomposition** is an iterative process of breaking the description or perspective of a system down into finer and finer detail. This process creates a set of hierarchically related charts in which one process on a given chart is explained in greater detail on another chart. For the Hoosier Burger system, we broke down, or decomposed, the larger system into four processes.

Each resulting process (or subsystem) is also a candidate for decomposition. Each process may consist of several sub processes. Each sub process may also be broken down into smaller units. Decomposition continues until you have reached the point at which no sub process can logically be broken down any further. The lowest level of a DFD is called a **primitive DFD**.

Rule	Incorrect	Correct
A.		
B.		
D.		
E.		
F.		
H.		
J.		

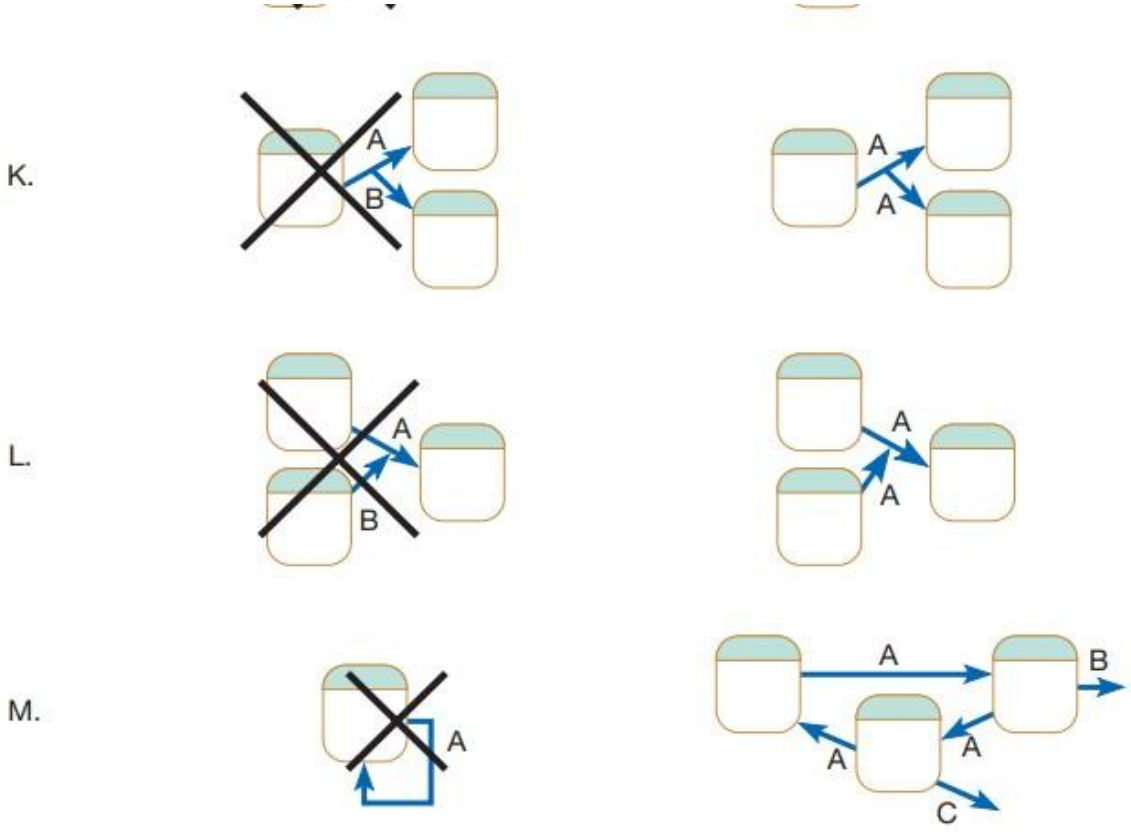


FIGURE 7-6

Incorrect and correct ways to draw DFDs

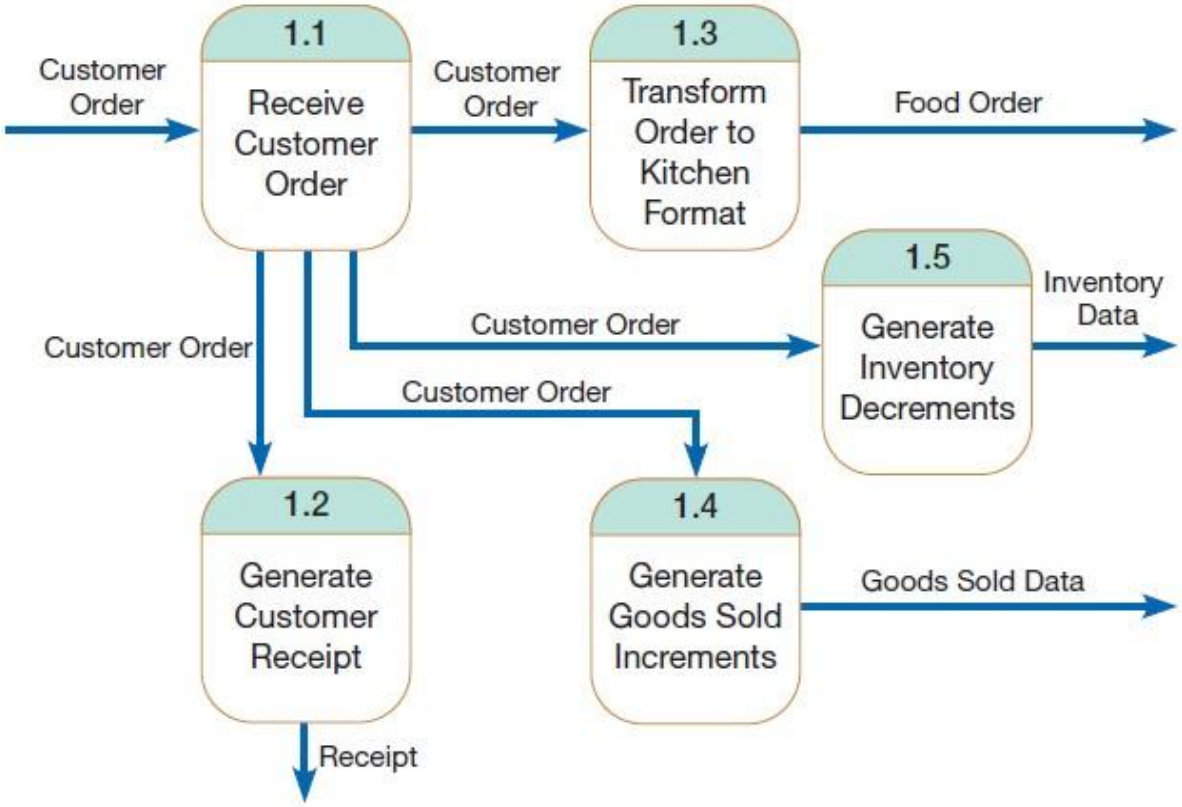


FIGURE 7-7
Level-1 diagram showing the decomposition of Process 1.0 from the level-0 diagram for Hoosier Burger's food-ordering system

Note that each of the five Processes in Figure 7-7 is labeled as a sub process of process 1.0: Process 1.1, Process 1.2, and so on. Also note that, just as with the other DFDs we have looked at, each of the processes and data flows is named. You will also notice that no sources or sinks are represented. Although you may include sources and sinks, the context and level-0 diagrams show the sources and sinks. The DFD in **Figure 7-7 is called a level-1 diagram**. If we should decide to decompose Processes 2.0, 3.0, or 4.0 in a similar manner, the DFDs we would create would also be level-1 diagrams. In general, a **level-n diagram is a DFD that is generated from n nested decompositions** from a level-0 diagram.

Processes 2.0 and 3.0 perform similar functions in that they both use data input to update data stores. Because updating a data store is a singular logical function, neither of these processes needs to be decomposed further. We can, however, decompose Process 4.0, Produce Management Reports, into at least three sub processes:**Access Goods Sold and Inventory Data**, **Aggregate Goods Sold and Inventory Data,** and **Prepare Management Reports**. The decomposition of Process 4.0 is shown in the level-1 diagram of Figure 7-8.

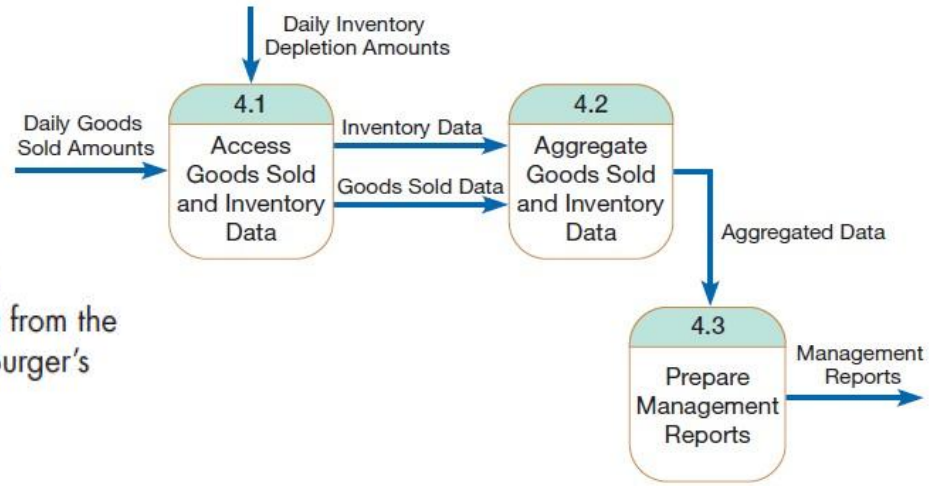


FIGURE 7-8
Level-1 diagram showing the decomposition of Process 4.0 from the level-0 diagram for Hoosier Burger's food-ordering system

Each level-1, -2, or -n DFD represents one process on a level-n-1 DFD;each DFD should be on a separate page.

No DFD should have more than about **seven processes** because too many processes will make the diagram **too crowded and difficult** to understand. Typically, **process names** begin with an **action verb**, such as **Receive**, **Calculate**,

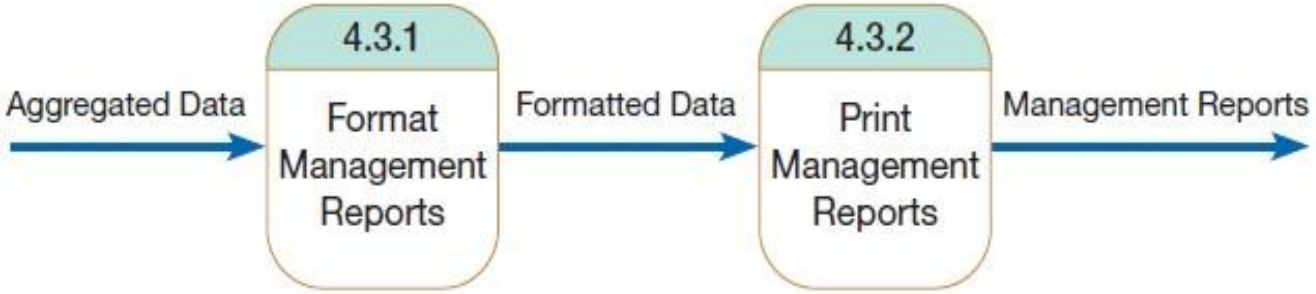


FIGURE 7-9
Level-2 diagram showing the decomposition of Process 4.3 from the level-1 diagram for Process 4.0 for Hoosier Burger's food-ordering system

Transform, **Generate**, or **Produce**. Process names often are the **same as the verbs used in many computer programming languages**. Example process names include Merge, Sort, Read, Write, and Print.

Balancing DFDs

When you decompose a DFD from one level to the next, there is a conservation principle at work. You must conserve inputs and outputs to a process at the next level of decomposition. In other words, **Process 1.0, which appears in a level-0 diagram, must have the same inputs and outputs when decomposed into a level-1 diagram**. This conservation of inputs and outputs is called **balancing.**

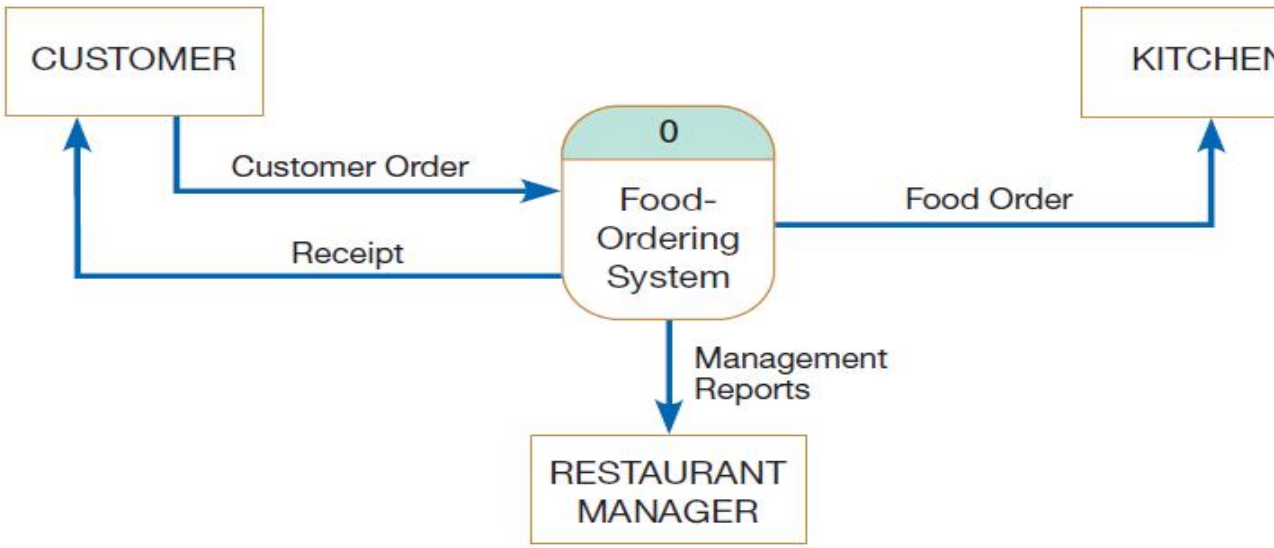


FIGURE 7-4
Context diagram of Hoosier Burger's food-ordering system

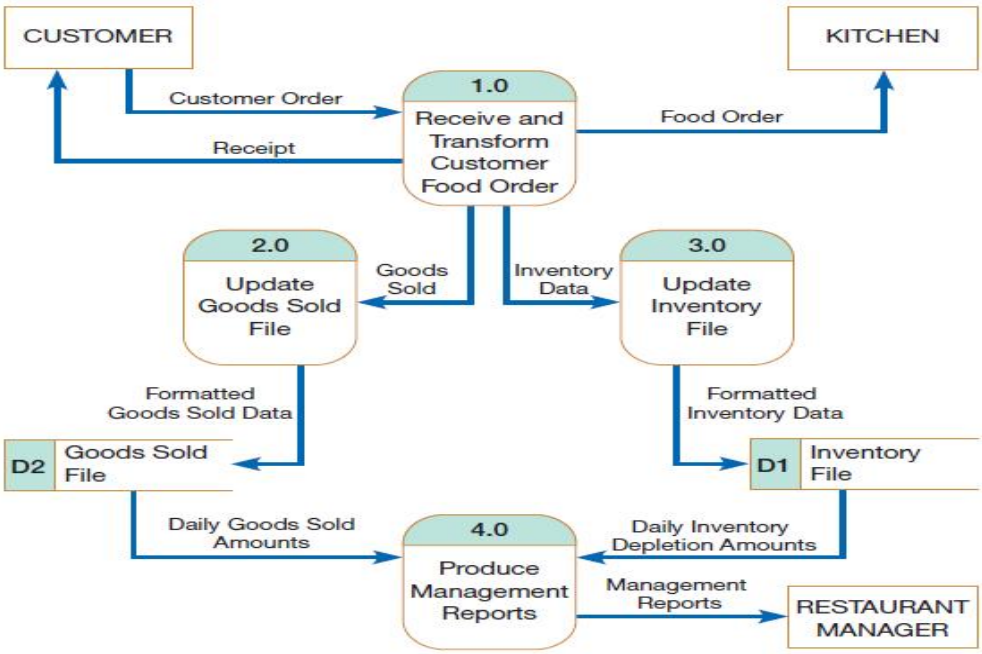


FIGURE 7-5
Level-0 DFD of Hoosier Burger's food-ordering system

Let's look at an example of balancing a set of DFDs. Look back at **Figure 7-4**. This is the context diagram for Hoosier Burger's food-ordering system. Notice that there is one input to the system, the customer order, which originates with the customer. Notice also that there are three outputs: the customer receipt, the food order intended for the kitchen, and management reports. Now look at **Figure 7-5**. This is the level-0 diagram for the food-ordering system. Remember that all data stores and flows to or from them are internal to the system. Notice that the same single input to the system and the same three outputs represented in the context diagram also appear at level 0. Further, no new inputs to or outputs from the system have been introduced. Therefore, we can say that the context diagram and level-0 DFDs are balanced.

Using Data flow Diagramming in the Analysis Process

Learning the mechanics of drawing DFDs is important because DFDs haveproven to be essential tools for the structured analysis process. Beyond the issue of drawing mechanically correct DFDs, there are other issues related to **process modeling** with which an analyst must be concerned. Such issues, including

whether the DFDs are **complete** and **consistent across all levels**, which covers **guidelines for drawing DFDs**. Another issue to consider is how you can use DFDs as a useful tool for analysis.

Guidelines for drawing DFDs

1. Completeness : The concept of **DFD completeness refers to whether you have included** in your DFDs all of the components necessary for the system you are modeling. If your DFD contains **data flows** that do not lead anywhere or data **stores, processes, or external entities** that are not connected to anything else, your DFD is not complete.

2. Consistency : The concept of **DFD consistency refers to whether ornot the depiction** of the system shown at one level of a nested set of DFDs is compatible with the depictions of the system shown at other levels. **A gross violation of consistency would be a level-1 diagram with no level-0 diagram. Another example of inconsistency would be a data flow that appears on a higher-level DFD but not on lower levels** (also a violation of balancing).

3. Timing: You may have noticed in some of the DFD examples we have presented that DFDs do not do a very good job of representing time. Ona given DFD, there is no indication of whether a data flow occurs constantly in real time, once per week, or once per year. There is also no indication of when a system would run.

4. Iterative Development: The first DFD you draw will rarely capture perfectly the system you are modeling. You should count on drawing the same

diagram over and over again, in an iterative fashion. With each attempt, you will come closer to a good approximation of the system or aspect of the system you are modeling. One rule of thumb is that it should take you about three revisions for each DFD you draw.

5. Primitive DFDs : One of the more difficult decisions you need to make when drawing DFDs is when to stop decomposing processes. One rule is to stop drawing when you have reached the lowest logical level; however, it is not always easy to know what the lowest logical level is.

Modeling logic with decision Tables

Decesion Tables:-

These are the tools required for the Structured approach. These tools are used to define clearly and concisely the decesion Statement of any problem in the tabulear form. These tools are considered as the powerful tool for defining the complex program logic. When the computer has to solve a problem having the large no. of results and to find the decesion among them or If there are number of different branches within any program the decesion tables are used. These tables tries to display the information clearly at a glance. The structure of decesion table is as follows

Condition Stub	Rules Entry
Action Stub	Action Entry

Example

In any computer installation company a candidate is interviewing for the job. The questions are as of the form. Is he sharp? Is he disciplined? & There two questions (conditions) lead the four possible combination

- Is she sharp & disciplined?
- Is only sharp?
- Is only disciplined?
- Is neither.

Each of above will result in a different activity

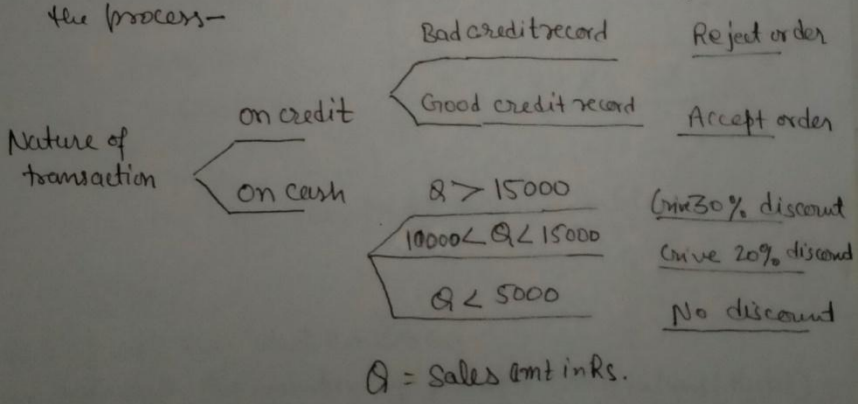
* Take him as a programmer.
* Take him as a trainee programmer
* Take him as a trainee operator
* Don't take him.

The decision table has the following format.

	Conditions	Rules			
		1	2	3	4
1.	Is sharp?	Y	Y	N	N
2.	Is disciplined	Y	N	Y	N
	Action	Action Entry			
1.	Appoint as programmer	X			
2.	Appoint as trainee prog		X		
3	Appoint as trainee operator			X	
4	Don't appoint				X

Decision Tree:-

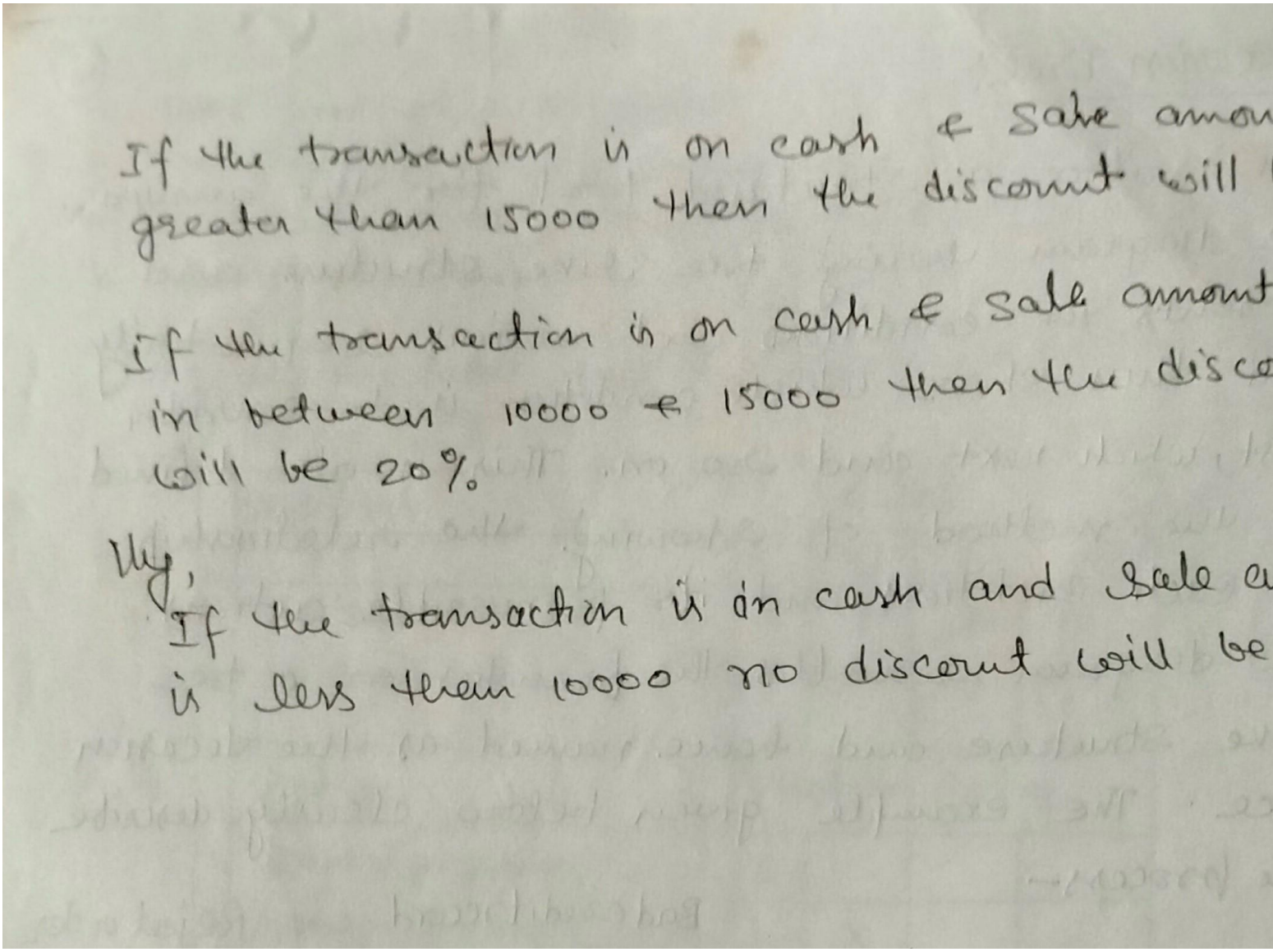
This is also a structured tool for the analysis. The diagram having tree like structure and presents the conditions and actions sequentially and thus shows which condition is to consider first, which next and so on. This is also defined as the method of showing the relationship of each conditions and its permissible actions. The diagram resembles the branches on a tree like structure and hence named as the decision tree. The example given below clearly describe the process-



Q = Sales amt in Rs.

Here,

* If the transaction is on credit and customer credit record is good then only the order is accepted and if the customer credit record is bad the order is rejected.



A decision table is a diagram of process logic where the logic is reasonably complicated. All of the possible **choices and the conditions** the choices depend on are represented in tabular form, as illustrated in the decision table in **Figure 7-18**. The decision table in Figure 7-18 models the logic of a generic payroll system.

The table has three parts: the **condition** stubs**, the action** stubs**, and the rules.** The condition stubs contain the various conditions that apply to the situation the table is modeling. In

Figure 7-18, there are two condition stubs for **employee type** and **hours worked**.

Employee type has two values: **"S,"** which stands for **salaried**, and **"H,"** which stands for **hourly**. Hours worked has three values: **less than 40**, **exactly 40**, and **more than 40**.

The action stubs contain all the possible courses of action that result from combining values of the condition stubs. There are four possible courses of action in this table: **Pay Base Salary**, Calculate **Hourly Wage**, Calculate **Overtime**, and **Produce Absence Report**. You can see that not all actions are triggered by all combinations of conditions. Instead, specific combinations trigger specific actions. The part of the table that links conditions to actions is the section that contains the rules.

	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce absence report		X				

FIGURE 7-18
Complete decision table for payroll system example

Conditions/ Courses of Action	Rules		
	1	2	3
Employee type	S	H	H
Hours worked	–	< 40	40
Pay base salary	X		
Calculate hourly wage		X	X
Calculate overtime			
Produce absence report		X	

FIGURE 7-19
Reduced decision table for payroll system example

To read the rules, start by reading the values of the conditions as specified in the first column: Employee type is "S," or salaried, and hours worked is less than 40. When both of these conditions occur, the payroll system is to pay the base salary.

In the next column, the values are "H" and "<40," meaning an hourly worker who worked less than 40 hours. In such a situation, the payroll system calculates the hourly wage and makes an entry in the Absence Report. Rule 3 addresses the situation when a salaried employee works exactly 40 hours. The system pays the base salary, as was the case for rule 1. For an hourly worker who has worked exactly 40 hours, rule 4 calculates the hourly wage. Rule 5 pays the base salary for salaried

employees who work more than 40 hours. Rule 5 has the same action as rules 1 and 3 and governs behavior with regard to salaried employees. The number of hours worked does not affect the outcome for rules 1, 3, or 5. For these rules, hours worked is an **indifferent condition in that its value does not affect the action taken.** Rule 6 calculates hourly pay and overtime for an hourly worker who has worked more than 40 hours.

indifferent condition

- In a decision table, a condition whose value does not affect which actions are taken for two or more rules.

Conditions/ Courses of Action	Rules			
	1	2	3	4
Employee type	S	H	H	H
Hours worked	–	<40	40	>40
Pay base salary	X			
Calculate hourly wage		X	X	X
Calculate overtime				X
Produce absence report		X		

FIGURE 7-19
Reduced decision table for payroll system example

Because of the indifferent condition for rules 1, 3, and 5, we can redu
he number of rules by **condensing rules 1, 3, and 5 into one rule**,
hown in Figure 7-19. **The indifferent condition is represented w**
dash. Whereas we started with a decision table with six rules,
ow have a simpler table that conveys the same information with o
our rules.

Decision Tree:

Decision Trees are considered to be one of the most popular approaches for representing classifiers. Researchers from various disciplines such as statistics, ma- chine learning, pattern recognition, and Data Mining have dealt with the issue of growing a decision tree from available data. This paper presents an updated survey of current methods for constructing decision tree classifiers in a top-down manner. The chapter suggests a unified algorithmic framework for presenting these algorithms and describes various splitting criteria and pruning methodologies.

A decision tree is a classifier expressed as a recursive partition of the instance space. The decision tree **consists of nodes that form a *rooted tree*, meaning it is a *directed tree* with a node called "root" that has no incoming edges**. All other nodes have exactly one incoming edge. A node with outgoing edges is called an *internal* or test node. All other nodes are called leaves (also known as terminal or decision nodes). In a decision tree, each internal node splits the instance space into two or more sub-spaces according to a certain discrete function of the input attributes values. In the simplest and most frequent case, each test considers a single attribute, such that the instance space is partitioned according to the attribute's value. In the case of numeric attributes, the condition refers to a range. Each leaf is assigned to one class representing the most appropriate target value. Alternatively, the leaf may hold a probability vector indicating the prob- ability of the

target attribute having a certain value. Instances are classified by navigating them from the root of the tree down to a leaf, according to the outcome of the tests along the path. Figure 9.1 describes a decision tree that reasons whether or not a potential customer will respond to a direct mailing. Internal nodes are represented as circles, whereas leaves are denoted as tri- angles. Note that this decision tree incorporates both nominal and numeric at- tributes. Given this classifier, the analyst can predict the response of a potential customer (by sorting it down the tree), and understand the behavioral charac- teristics of the entire potential customers population regarding direct mailing. Each node is labeled with the attribute it tests, and its branches are labeled with its corresponding values.

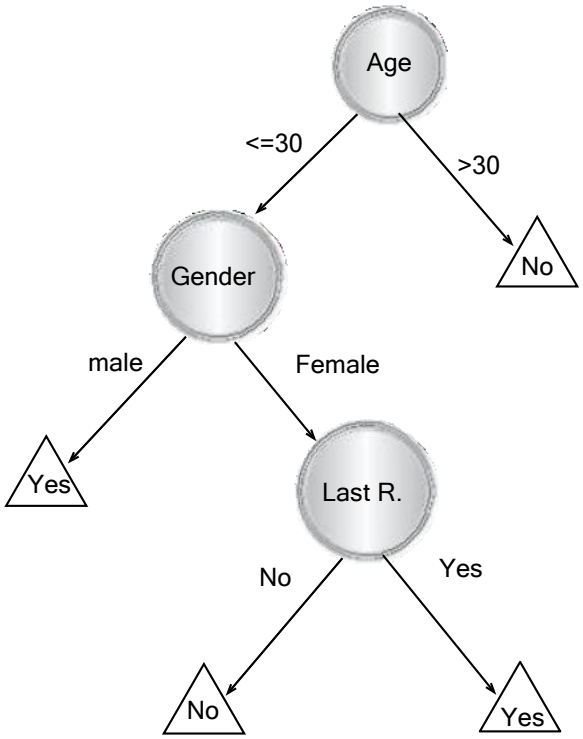


Figure 9.1. Decision Tree Presenting Response to Direct Mailing.

In case of numeric attributes, decision trees can be geometrically interpreted as a collection of hyperplanes, each orthogonal to one of the axes. Naturally, decision-makers prefer less complex decision trees, since they may be consid- ered more comprehensible. Furthermore, according to Breiman *et al*. (1984) the tree complexity has a crucial effect on its accuracy. The tree complexity is explicitly controlled by the stopping criteria used and the pruning method employed. Usually the tree complexity is measured by one of the following metrics: the total number of nodes, total number of leaves, tree depth and number of attributes used. Decision tree induction is closely related to rule induction. Each path from the root of a decision tree to one of its leaves can be transformed into a rule simply by conjoining the tests along the path to form the antecedent part, and taking the leaf's class prediction as the class value. For example, one of the paths in Figure 9.1 can be transformed into the rule: "If customer age is is less than or equal to or equal to 30, and the gender of the customer is "Male" – then the customer will respond to the mail". The resulting rule set can then be simplified to improve its comprehensibility to a human user, and possibly its accuracy (Quinlan, 1987).

Pseudocode:

It may be that when we've done a careful job of articulating an algorithm, the result is so precise, so unambiguous, and so clearly structured that a computer implementation of the algorithm can be written with little or no additional preparation—even by a programmer who has no prior familiarity with the algorithm in question. In this case, we might say that our articulation of the algorithm is actually a kind of *pseudocode*: it has many characteristics in common with programming language code, and it may appear very much like such code, but it is not, in fact, directly usable as programming language code.

Pseudocode is a very useful device for specifying the logic of a computer program (or some critical portion of a program) prior to that program

actually being written, as well as for documenting the logic of a computer program after the fact. It can be used to express the high- level logic of a traditional program, the lower-level details of a core function in an operating system or run-time library, the behaviors and methods in an agent-based or object-oriented program, and everything in between. But as useful as pseudocode is, there's a catch: unlike actual programming languages, and unlike natural languages, there's no standard vocabulary or grammar for pseudocode. Pseudocode can be expressed in virtually any written language in existence. It can look very much like standard (even if very formal) prose; at the other end of the spectrum, it can appear so close to programming code that on first glance, we might think that's exactly what it is. So what is pseudocode? One way to describe it is easy, but arguably not very useful: in practice, **pseudocode is simply a very precise, minimally ambiguous articulation of an algorithm**—but even more precise, and less ambiguous, than usual. **Another clue comes from the world of mathematics: pseudocode often employs algebraic variable naming and expressions, as well as notation from set theory, linear algebra, and other branches of mathematics**. The use of these mathematical conventions can go a long way toward eliminating, or at least reducing, ambiguity in the description of an algorithm. Ultimately, the most important characteristic of pseudocode is not really what it *is*, but what it *makes possible.* As noted above, when we start with well-written pseudocode, virtually any programmer with reasonable competence in a given programming language should be able to implement the algorithm described by the pseudocode, in the given language, with little or no need for further instruction.

Writing pseudocode

Given the fact that there isn't a standard pseudocode language, we're mostly left to our own devices to come up with a suitable grammar and

vocabulary for the pseudocode we write (unless, of course, we happen to be working in or for an organization which has well-established standards or conventions for pseudocode). But others have gone before us, and we can learn from them.

Pseudocode guidelines

1. Avoid mixing and matching of natural languages (just as you should when naming variables and methods in program code). For example, if you're writing pseudocode in English, avoid including terms or variable names from other languages, unless there's a compelling reason to do so.
2. Strive for consistency, unless doing so would make the pseudocode less clear. For example, if in one part of your pseudocode you use a particular symbol or verb to denote assignment of a value to a variable, use that same symbol or verb throughout.
3. Where a step in the algorithm is expressed primarily in natural language, with few symbolic notations, use proper grammar. However, remember that mathematical expressions are often less ambiguous than natural language, even with correct grammar.
4. Since most programming languages borrow keywords from English, it's to be expected that pseudocode will resemble programming code to some extent. However, pseudocode should not be tightly coupled with any single programming language. Instead, it should employ control structures, verbs, and other keywords that are common to most programming languages. For example, most imperative programming languages have *if-then-else*, *for-next*, and *while* flow control statements; when combined with mathematical symbols for calculation of simple expressions and assignment of values to variables, these are sufficient for expressing virtually any algorithm.
5. It isn't necessary (or even desirable, in many cases) to have a one-to-one correspondence between each line of pseudocode and a corresponding line of program code, or between the symbolic names used in pseudocode and those used in program code. In particular, many common high-level operations (e.g. sorting values, searching for a minimum or maximum value from a list, opening a file to read a value from it) should generally be stated in a single line of pseudocode, rather than including all of the steps necessary to perform those operations in practice—unless, of course, the algorithm being described is for performing just such an operation.
6. Use indentation to make any non-linear structure apparent. For example, when using an *if-then* statement to show that some portion of the algorithm should be performed conditionally, place the conditional portion immediately below the *if-then* statement, and indent it one tab stop to the right of the *if-then* statement itself. Similarly, if some portion of the algorithm is to be performed iteratively, under the control of a *for-next* or

while statement, place that portion of the algorithm immediately below the *for-next* or *while* statement, and indent it one tab stop further to the right. (By the way, these are good indentation practices for program code as well—even required in some cases.)

7. If an algorithm is so long or complex that the pseudocode becomes hard to follow, try breaking it up into smaller, cohesive sections, each with its own title; we can think of these sections as the pseudocode analogues to methods, functions, and procedures. When you do this, make sure that the pseudocode also includes an articulation of the higher-level sequence, showing the order in which the more detailed sections should beperformed.

Pseudo-code is an informal way to express the design of a computer program or an algorithm in 1.45. The aim is to get the idea quickly and also easy to read without details. It is like a young child putting sentences together without any grammar. There are several ways of writing pseudo-code; there are no strict rules. But to reduce ambiguity between what you are required to do and what you express let's base the pseudo code on the few defined conventions and carry out the exercises.

Pseudo-code Examples

Repeatedly steps through the list to

Let's see few examples that can be used to write pseudo-code.

1. **Sort**

Taking the sorting example; let's sort an array using the ***Bubble sort technique***. This sorting algorithm could be implemented in all programming languages but let's see the C implementation.

```
void ArraySort(int This[], CMPFUN fun_ptr, uint32 ub)
{
 /* bubble sort */
```

```
 uint32 indx; uint32
 indx2;int temp;
int temp2; int
flipped;

if (ub <= 1)return;

 indx = 1;do
        {
   flipped = 0;
    for (indx2 = ub - 1; indx2 >= indx; --indx2)
                {
temp = This[indx2];
 temp2 = This[indx2 - 1];
if ((*fun_ptr)(temp2, temp) > 0)
                {
This[indx2 - 1] = temp;This[indx2]
= temp2;
 flipped = 1;
                }
            }
        } while ((++indx < ub) && flipped);
      }
```

What's your impression?

Is it easy to understd at once this C implementation?

Bubble sort is mostly used in

Here is some pseudo code for this algorithm.

```
Set n to number of records
 to be sorted repeat
flag = false;
for counter = 1 to n-1 do
        if key[counter] >
        key[counter+1] then swap
        the records;
         set flag =
        true; end if
end do n = n-1;
until flag = false or n=1
```

What's easier to understand, the implementation in C or pseudo-code?