Chapter 9

From Structured Analysis Wiki

Dataflow Diagrams

"Form ever follows function."

-- Louis Henri Sullivan (http://en.wikipedia.org/wiki/Louis_Sullivan)
"The Tall Office Building Artistically Considered," (http://www.njit.edu/v2/Library/archlib/pub-domain/sullivan-1896-tall-bldg.html) *Lippincott's Magazine,* March 1896

Contents

INTRODUCTION

IN THIS CHAPTER, YOU WILL LEARN:

1. The components of a dataflow diagram (http://en.wikipedia.org/wiki/Dfd) ;
2. How to draw a simple dataflow diagram;
3. Guidelines for drawing successful dataflow diagrams; and
4. How to draw leveled dataflow diagrams.

In this chapter, we will explore one of the three major graphical modeling tools of structured analysis: the *dataflow diagram*. The dataflow diagram is a modeling tool that allows us to picture a system as a network of functional processes, connected to one another by "pipelines" and "holding tanks" of data. In the computer literature, and in your conversations with other systems analysts and business users, you may use any of the following terms as synonyms for dataflow diagram:

- Bubble chart
- DFD (the abbreviation we will use throughout this book)
- Bubble diagram
- Process model (or business process model)
- Business flow model
- Work flow diagram
- Function model
- "A picture of what's going on around here"

The dataflow diagram is one of the most commonly used systems-modeling tools, particularly for operational systems in which the *functions* of the system are of paramount importance and more complex than the data that the system manipulates. DFDs were first used in the software engineering field as a notation for studying systems design issues (e.g., in early structured design books and articles such as (Stevens, Myers, and Constantine (http://www.research.ibm.com/journal/sj/132/ibmsj1302C.pdf) . 1974), (Yourdon and Constantine (http://www.amazon.com/exec/obidos/ASIN/0138544719/edyourdonswebsit) , 1975), (Myers, 1975), et al.). In turn, the notation had been borrowed from earlier papers on graph theory, and it continues to be used as a convenient notation by software engineers concerned with direct implementation of models of user requirements.

This is interesting background, but is likely to be irrelevant to the users to whom you show DFD system models; indeed, probably the *worst* thing you can do is say, "Mr. User, I'd like to show you a top-down, partitioned, graph-theoretic model of your system." Actually, many users will be familiar with the underlying concept of DFDs, because the same kind of notation has been used by operations research scientists for nearly a century to build work-flow models of organizations. This is important to keep in mind: DFDs can be used not only to model information-processing systems, but also as a way of modeling whole organizations, that is, as a tool for business planning and strategic planning.

We will begin our study of dataflow diagrams by examining the components of a typical dataflow diagram: the process, the flow, the store, and the terminator. We will use a fairly standard notation for DFDs, following the notation of such classic books as (DeMarco (http://www.amazon.com/exec/obidos/ASIN/0138543801/edyourdonswebsit) , 1978), (Gane and Sarson (http://www.amazon.com/exec/obidos/ASIN/0930196007/edyourdonswebsit) , 1977), and others. However, we will also include DFD notation for modeling real-time systems (i.e., control flows and control processes). This additional notation is generally not required for business-oriented systems, but is crucial when modeling a variety of engineering and scientific systems.

Next, we will review some guidelines for constructing dataflow diagrams so that we can minimize the chances of constructing a confusing, incorrect, or inconsistent DFD. Finally, we will discuss the concept of leveled DFDs as a method of modeling complex systems.

Keep in mind that the DFD is just one of the modeling tools available to the systems analyst and that it provides only one view of a system — the function-oriented view. If we are developing a system in which data relationships are more important than functions, we might de-emphasize the DFD (or conceivably not even bother developing one) and concentrate instead on developing a set of entity-relationship diagrams as discussed in Chapter 12. Alternatively, if the time-dependent behavior of the system dominated all other issues, we might concentrate instead on the state-transition diagram discussed in Chapter 13.

THE COMPONENTS OF A DFD

Figure 9.1 shows a typical DFD for a small system. Before we examine its components in detail, notice several things:

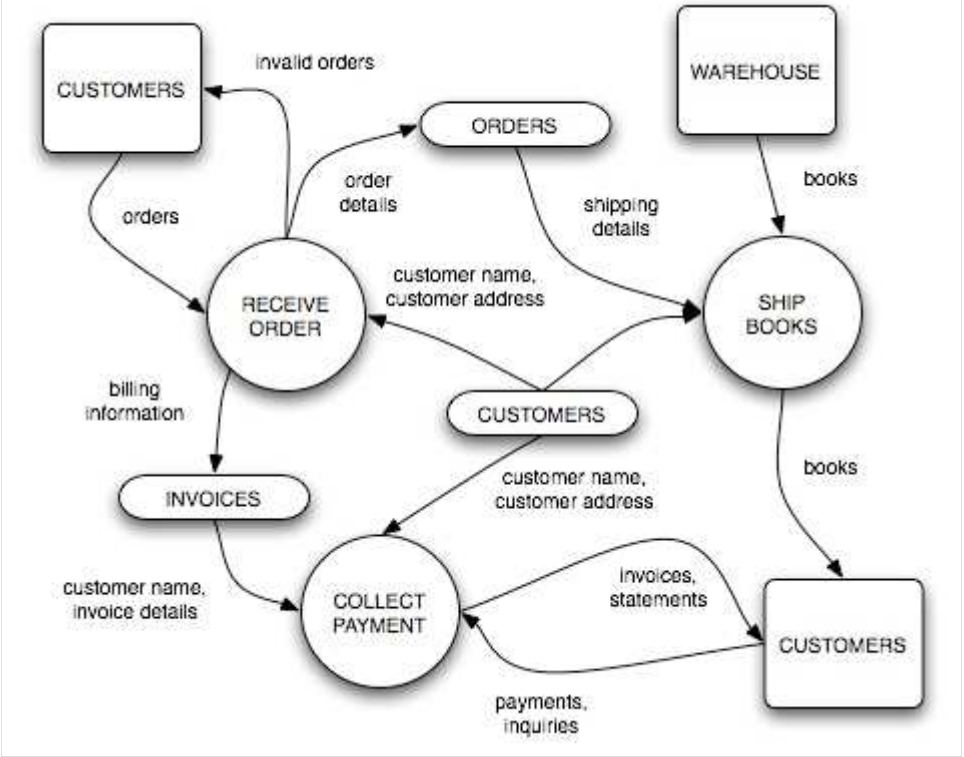


Figure 9.1: A typical DFD; source: Image:Figure91.graffle

- It hardly needs to be explained at all; one can simply look at the diagram and understand it. The notation is simple and unobtrusive and, in a sense, intuitively obvious. This is particularly important when we remember who is supposed to be looking at Figure 9.1 — not the systems analyst, but the user! If the user needs an encyclopedia in order to read and understand the model of his system, he or she probably won't bother to do either.
- The diagram fits easily onto one page. This means two things: (1) someone can look at the diagram without being overwhelmed, and (2) the system that is being modeled by the diagram is not very complex. What do we do if the system is intrinsically complex, for example, so complex that there would be literally hundreds of circles and lines in the diagram? We will discuss this in Section 9.4.
- The diagram has been drawn by a computer. There is nothing wrong with a hand-drawn diagram, but Figure 9.1 and many of the other DFDs shown in this book were originally drawn with the assistance of a simple Macintosh program called MacDraw. This means that the diagram is likely to be drawn more neatly and in a more standardized fashion than would normally be possible in a hand-drawn diagram. It also means that changes can be made and new versions produced in a matter of minutes.[1]

The Process

The first component of the DFD is known as a *process*. Common synonyms are a bubble, a function, or a transformation. The process shows a part of the system that transforms inputs into outputs; that is, it shows how one or more inputs are changed into outputs. The process is represented graphically as a circle, as shown in Figure 9.2(a). Some systems analysts prefer to use an oval or a rectangle with rounded edges, as shown in Figure 9.2(b); still others prefer to use a rectangle, as shown in Figure 9.2(c). The differences between these three shapes are purely cosmetic, though it is obviously important to use the same shape consistently to represent all the functions in the system. Throughout the rest of this book, we will use the circle or bubble.[2]

Figure 9.2(a): An example of a process; source: Image:Figure92a.graffle

Figure 9.2(b): An alternative representation of a process; source: Image:Figure92b.graffle

Figure 9.2(c): Still another representation of a process; source: Image:Figure92c.graffle

Note that the process is named or described with a single word, phrase, or simple sentence. For most of the DFD models that we will discuss in this book, the process name will describe *what* the process does. In Section 9.2, we will say more about proper naming of process bubbles; for now, it is sufficient to say that a good name will generally consist of a verb-object phrase such as **VALIDATE INPUT** or **COMPUTE TAX RATE**.

In some cases, the process will contain the name of a person or a group of people (e.g., a department or a division of an organization), or a computer, or a mechanical device. That is, the process sometimes describes who or what is carrying out the process, rather than describing what the process is. We will discuss this in more detail in Chapter 17 when we discuss the concept of an *essential* model, and later in Part IV when we look at implementation models.

The Flow

A *flow* is represented graphically by an arrow into or out of a process; an example of flow is shown in Figure 9.3. The flow is used to describe the movement of chunks, or packets of information from one part of the system to another part. Thus, the flows represent data in motion, whereas the stores (described below in Section 9.1.3) represent data at rest.

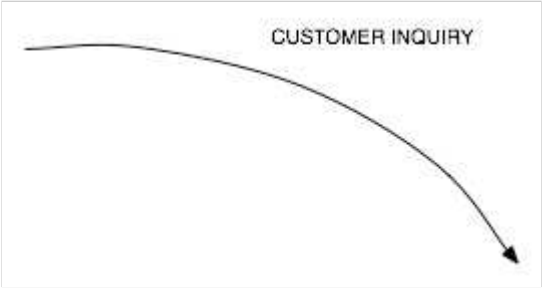


Figure 9.3: An example of a flow; source: Image:Figure93.graffle

For most of the systems that you model as a systems analyst, the flows will indeed represent data, that is, bits, characters, messages, floating point numbers, and the various other kinds of information that computers can deal with. But DFDs can also be used to model systems other than automated, computerized systems; we may choose, for example, to use a DFD to model an assembly line in which there are no computerized components. In such a case, the packets or chunks carried by the flows will typically be physical materials; an example is shown in Figure 9.4. For many complex, real-world systems, the DFD will show the flow of materials and data.

The flows in Figures 9.3 and 9.4 are *named*. The name represents the meaning of the packet that moves along the flow. A corollary of this is that the flow carries only one type of packet, as indicated by the flow name. The systems analyst should not name a dataflow **APPLES AND ORANGES AND WIDGETS AND VARIOUS OTHER THINGS**. However, we will see in Part III, that there are exceptions to this convention: it is sometimes useful to consolidate several elementary dataflows into a consolidated flow. Thus, one might see a single dataflow labeled **VEGETABLES**

instead of several different dataflows labeled **POTATOES**, **BRUSSEL SPROUTS**, and **PEAS**. As we will see, this will require some explanation in the data dictionary, which is discussed in Chapter 10.

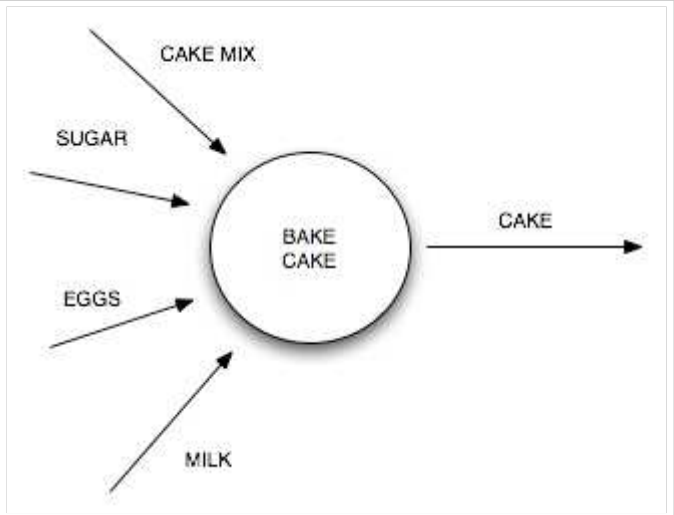


Figure 9.4: A DFD with material flows; source: Image:Figure94.graffle

While this may seem like an obvious point, keep in mind that the same content may have a different meaning in different parts of the system. For example, consider the fragment of a system shown in Figure 9.5.

The same chunk of data (e.g., 212-410-9955) has a different meaning when it travels along the flow labeled **PHONE-NUMBER** than it does when it travels along the flow labeled **VALID-PHONE-NUMBER**. In the first case, it means a telephone number that may or may not turn out to be valid; in the second case, it means a phone number that, within the context of this system, is known to be valid. Another way to think of it is that the flow labeled "phone number" is like a pipeline, undiscriminating enough to allow invalid phone numbers as well as valid phone numbers to travel along it; the flow labeled **VALID-PHONE-NUMBER** is narrower, or more discriminating, and allows a more narrowly defined set of data to move through it.

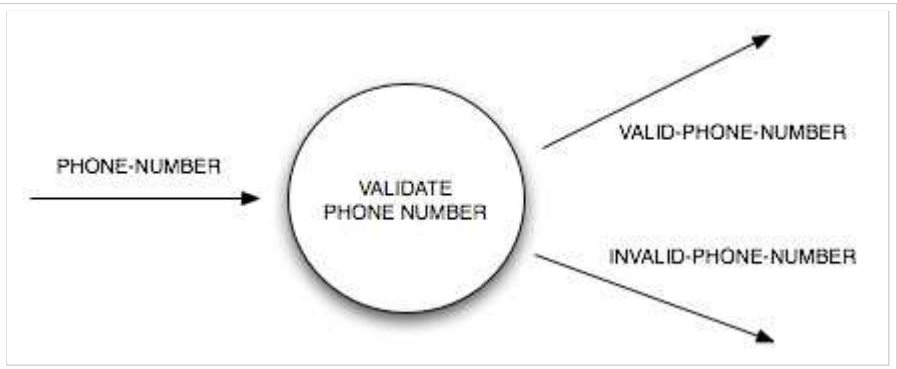


Figure 9.5: A typical DFD; source: Image:Figure95.graffle

Note also that the flows show direction: an arrowhead at either end of the flow (or possibly at both ends) indicates whether data (or material) are moving into or out of a process (or doing both). The flow shown in Figure 9.6(a), for example, clearly shows that a telephone number is being sent into the process labeled **VALIDATE PHONE NUMBER**. And the flow labeled **TRUCKER-DELIVERY-SCHEDULE** in Figure 9.6(b) is clearly an output flow generated by the process **GENERATE TRUCKER DELIVERY SCHEDULE**; data moving along that flow will either travel to another process (as an input) or to a store (as discussed in Section 9.1.3) or to a terminator (as discussed in Section 9.1.4). The double-headed flow shown in Figure 9.6(c) is a dialogue, a convenient packaging of two packets of data (an inquiry and response or a question and answer) on the same flow. In the case of a dialogue, the

packets at either end of the arrow must be named, as illustrated by Figure 9.6(c).[3]

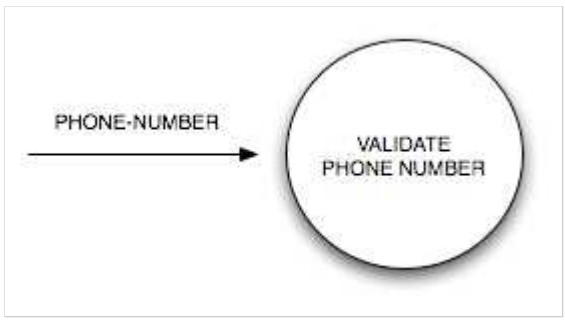


Figure 9.6(a): An input flow; source: Image:Figure96a.graffle

Dataflows can diverge and converge in a DFD; conceptually, this is somewhat like a major river splitting into smaller tributaries, or tributaries joining together. However, this has a special meaning in a typical DFD in which packets of data are moving through the system: in the case of a diverging flow, it means that duplicate copies of a packet of data are being sent to different parts of the system, or that a complex packet of data is being split into several more elementary data packets, each of which is being sent to different parts of the system, or that the dataflow pipeline carries items with different values (e.g., vegetables whose values may be "potato," "brussel sprout," or "lima bean") that are being separated. Conversely, in the case of a converging flow, it means that several elementary packets of data are joining together to form more complex, aggregate packets of data. For example, Figure 9.7(a) shows a DFD in which the flow labeled **ORDER-DETAILS** diverges and carries copies of the same packets to processes **GENERATE SHIPPING DOCUMENTS**, **UPDATE INVENTORY**, and **GENERATE INVOICE**. Figure 9.7(b) shows the flow labeled **CUSTOMER-ADDRESS** splitting into more elementary packets labeled **PHONE-NUMBER**, **ZIP-CODE**, and **STREET-ADDRESS**, which are sent to three different validation processes.[4]

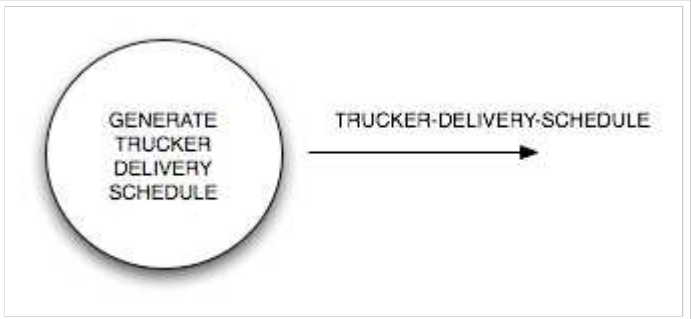


Figure 9.6(b): An output flow; source: Image:Figure96b.graffle

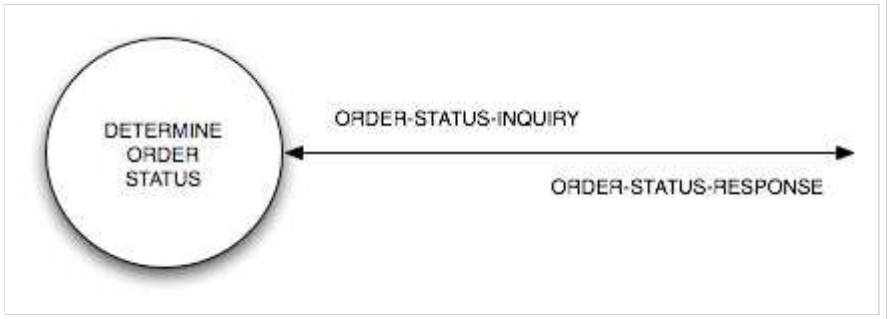


Figure 9.6(c): A dialog flow; source: Image:Figure96c.graffle

Note that the flow doesn't answer a lot of procedural questions that you might have when looking at the DFD: it doesn't answer questions about input prompts, for example, and it doesn't answer questions about output flows. For example, Figure 9.8(a) shows the simple case of an input flow coming into the process labeled **PROCESS ORDER**. But how does this happen? Does **PROCESS ORDER** explicitly ask for the input; for example, does it prompt the user of an on-line system, indicating that it wants some input? Or do data packets move along the flow of their own volition, unasked for? Similarly, Figure 9.8(b) shows a simple output flow emanating from '**GENERATE INVOICE REPORT**; do **INVOICE**s move along that flow when **GENERATE INVOICE REPORT** wants to send them, or when some other part of the system asks for the packet? Finally, consider the more common situation shown in Figure 9.8(c), in which there are multiple input flows and multiple output flows: in what sequence do the packets of data arrive, and in what *sequence* are the output packets generated? And is there a one-to-one ratio between the input packets and the output packets? That is, does process Q require exactly one packet from input flows A, B, and C in order to produce exactly one output packet for output flows X, Y, and Z? Or are there two As for every three Bs?

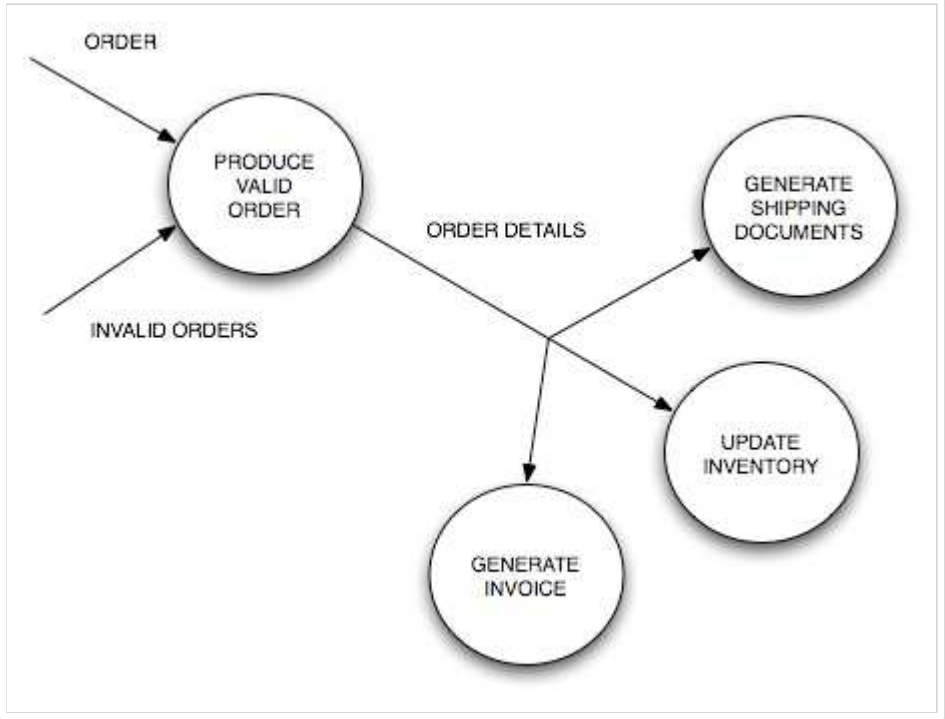


Figure 9.7(a): A diverging flow; source: Image:Figure97a.graffle

The answer to all these questions is very simple: we don't know. All these questions involve *procedural* details, the sort of questions that would normally be modeled with a flowchart or some other *procedural* modeling tool. *The DFD simply doesn't attempt to address such issues*. If these questions do become important to you, then you will have to model the internal procedure of the various processes; tools for doing this job are discussed in Chapter 11.

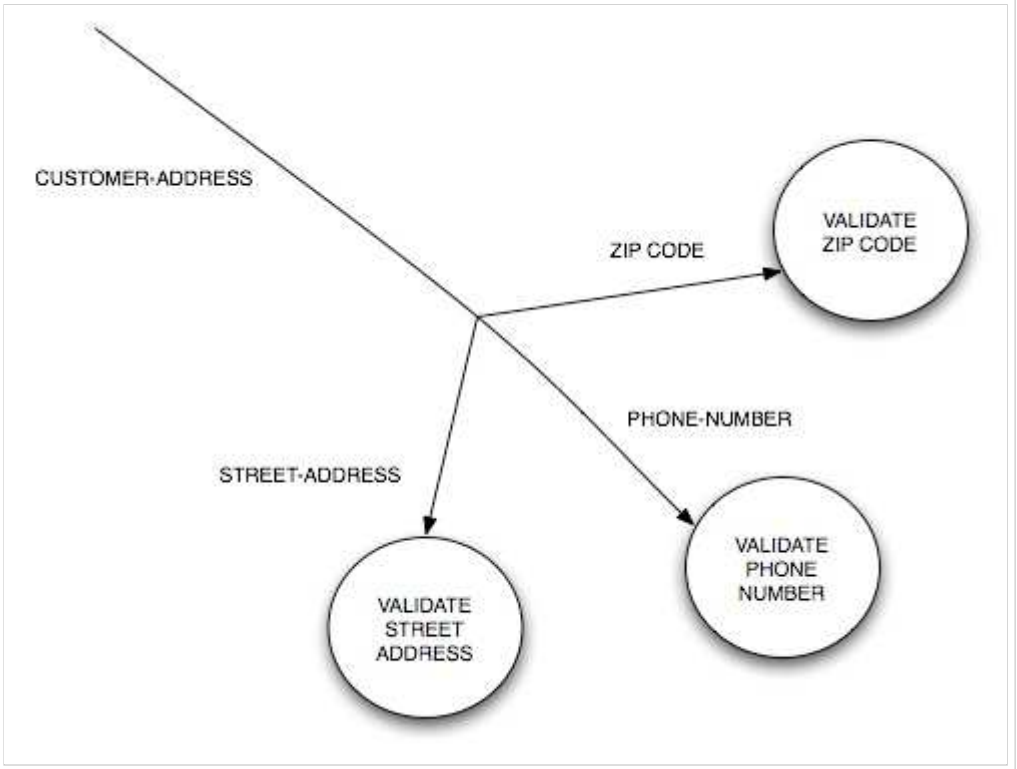


Figure 9.7(b): Another example of a diverging flow; source: Image:Figure97b.graffle

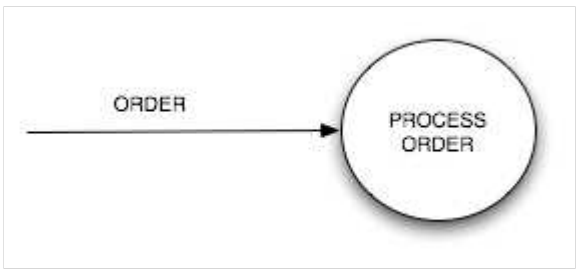


Figure 9.8(a): An input flow; source: Image:Figure98a.graffle

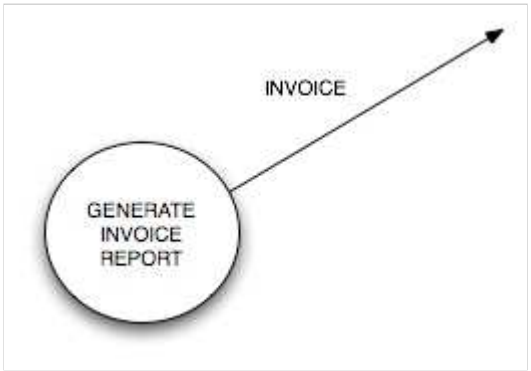


Figure 9.8(b): An output flow; source: Image:Figure98b.graffle

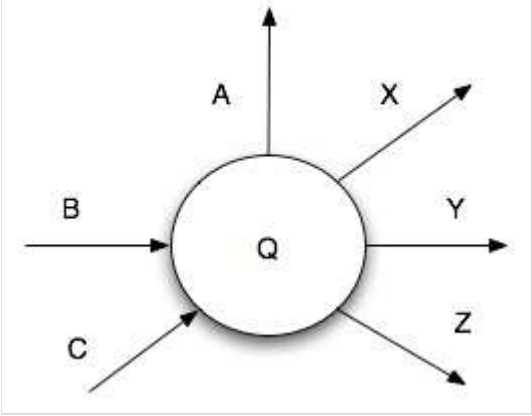


Figure 9.8(c): A combination of input and output flows; source: Image:Figure98c.graffle

The Store

The *store* is used to model a collection of data packets at rest. The notation for a store is two parallel lines, as shown in Figure 9.9(a); an alternative notation is shown in Figure 9.9(b)[5]; yet another notation, used in the case study in Appendix F, is shown in Figure 9.9(c). Typically, the name chosen to identify the store is the *plural* of the name of the packets that are carried by flows into and out of the store.

Figure 9.9(a): Graphical representation of a store; source: Image:Figure99a.graffle

Figure 9.9(b): An alternative notation for a store; source: Image:Figure99b.graffle

Figure 9.9(c): The notation used in Appendix F; source: Image:Figure99c.graffle

For the systems analyst with a data processing background, it is tempting to refer to the stores as *files* or *databases* (e.g., a disk file organized with Oracle, DB2, Sybase, Microsoft Access, or some other well-known database management system). Indeed, this is how stores are typically implemented in a computerized system; but a store can also be data stored on punched cards, microfilm, microfiche, or optical disk, or a variety of other electronic forms. And a store might also consist of 3-by-5 index cards in a card box, or names and addresses in an address book, or several file folders in a file cabinet, or a variety of other *non*-computerized forms. Figure 9.9(d) shows a typical example of a "store" in an existing *manual* system. It is precisely because of the variety of possible *implementations* of a store that we deliberately choose a simple, abstract graphical notation and the term store rather than, for instance, database.[6]

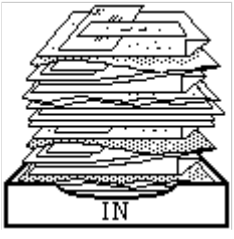


Figure 9.9(d): Another form of a store

Aside from the physical form that the store takes, there is also the question of its purpose: does the store exist because of a fundamental *user requirement*, or does it exist because of a convenient aspect of the *implementation* of the system? In the former case, the store exists as a necessary time-delayed storage area between two processes that occur at different times. For example, Figure 9.10 shows a fragment of a system in which, *as a matter of user policy* (independent of the technology that will be used to implement the system), the order entry process may operate at different times (or possibly at the same time) as the order inquiry process. The **ORDERS** store must exist in some form, whether on disk, tape, cards, or stone tablets.

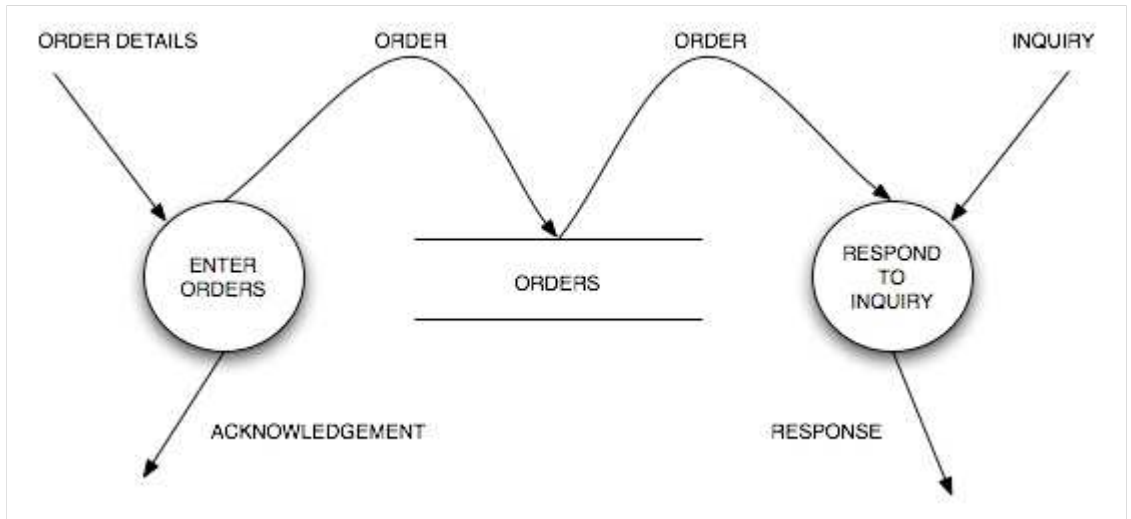


Figure 9.10: A necessary store; source: Image:Figure910.graffle

Figure 9.11(a) shows a different kind of store: the implementation store. We might imagine the systems designer interposing an **ORDERS** store between **ENTER ORDER** and **PROCESS ORDER** because:

- Both processes are expected to run on the same computer, but there isn't enough memory (or some other hardware resource) to fit both processes at the same time. Thus, the **ORDERS** store has been created as an intermediate file, because the available implementation technology has *forced* the processes to execute at different times.
- Either or both of the processes are expected to run on a computer hardware configuration that is somewhat unreliable. Thus, the **ORDERS** store has been created as a backup mechanism in case either process aborts.
- The two processes are expected to be implemented by different programmers (or, in the more extreme case, different *groups* of programmers working in different geographical locations). Thus, the **ORDERS** store has been created as a testing and debugging facility so that, if the entire system doesn't work, both groups can look at the contents of the store to see where the problem lies.
- The systems analyst or the systems designer thought that the user might eventually want to access the **ORDERS** store for some other purpose, even though the user did not indicate any such interest. In this case, the store has been created in anticipation of future user needs (and since it will cost something to implement the system in this fashion, the user will end up paying for a system capability that was not asked for).

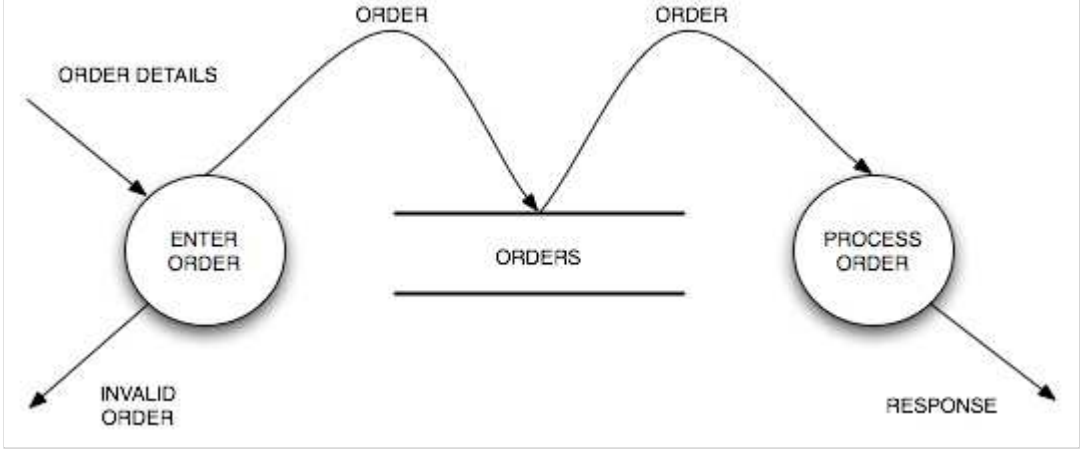


Figure 9.11(a): An "implementation" store; source: Image:Figure911a.graffle

If we were to exclude the issues and model only the *essential* requirements of the system, there would be no need for the **ORDERS** store; we would instead have a DFD like the one shown in Fig. 9.11(b).

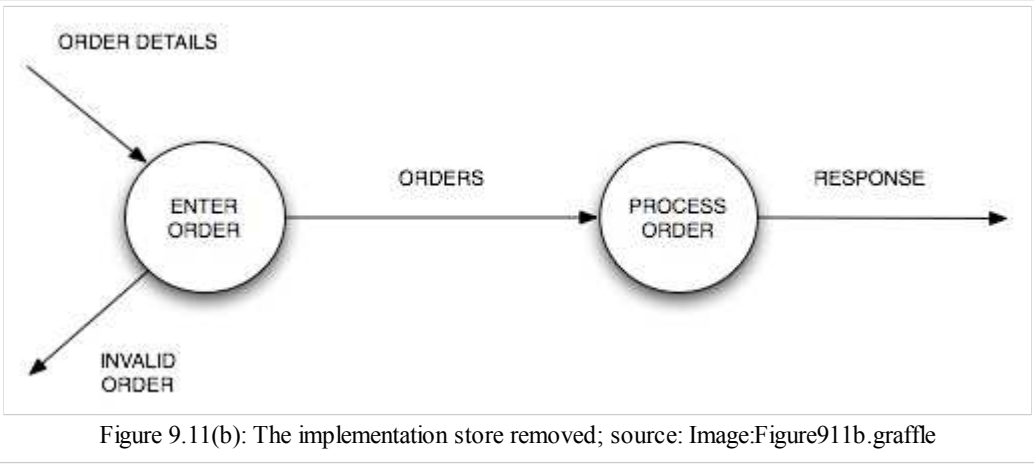


Figure 9.11(b): The implementation store removed; source: Image:Figure911b.graffle

As we have seen in the examples thus far, stores are connected by flows to processes. Thus, the context in which a store is shown in a DFD is one (or both) of the following:

- A flow from a store
- A flow to a store

In most cases, the flows will be labeled as discussed in Section 9.1.3. However, many systems analysts do not bother labeling the flow if an entire instance of a packet flows into or out of the store[7]. For example, Figure 9.12 shows a typical fragment of a DFD.

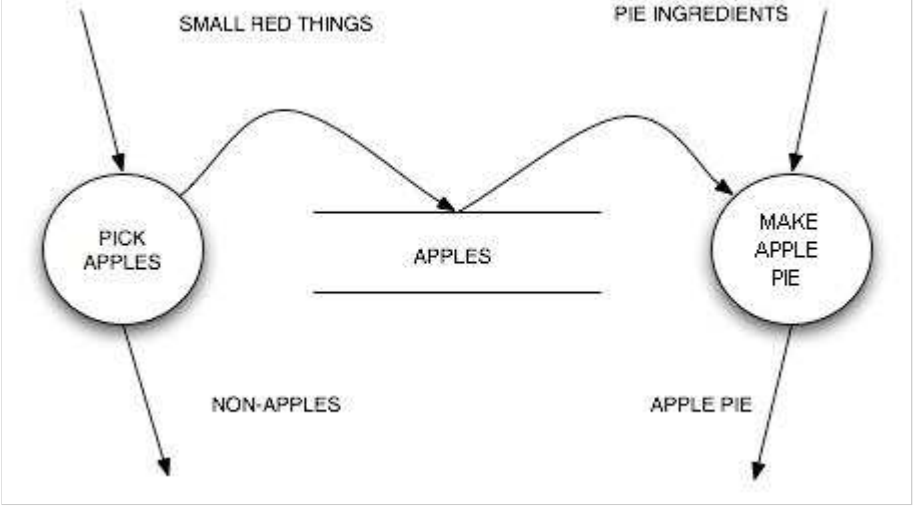


Figure 9.12: Stores with unlabeled flows; source: Image:Figure912.graffle

A flow from a store is normally interpreted as a read or an access to information in the store. Specifically, it can mean that:

- A single packet of data has been retrieved from the store; this is, in fact, the most common example of a flow from a store. Imagine, for example, a store called **CUSTOMERS**, where each packet contains name, address, and phone number information about individual customers. Thus, a typical flow from the store might involve the retrieval of a complete packet of information about one customer.
- More than one packet has been retrieved from the store. For example, the flow might retrieve packets of information about all the customers from New York City from the **CUSTOMERS** store.
- A portion of one packet from the store. In some cases, for example, only the phone number portion of information from one customer might be retrieved from the **CUSTOMERS** store.
- Portions of more than one packet from the store. For example, a flow might retrieve the zip-code portion of all customers living in the state of New York from the **CUSTOMERS'** store.

As we noted earlier when we examined flows entering and leaving a process, we will have many *procedural* questions; for example, does the flow represent a single packet, multiple packets, portions of a packet, or portions of several packets? In some cases, we can tell simply by looking at the label on the flow: if the flow is unlabeled, it means that an entire packet of information is being retrieved (as indicated above, this is simply a convenient convention); if the label on the flow is the same as that of the store, then an entire packet (or multiple instances of an entire packet) is being retrieved; and if the label on the flow is something other than the name of the store, then one or more components of one or more packets are being retrieved.[8]

While some of the procedural questions can thus be answered by looking carefully at the labels attached to a flow, not all the details will be evident. Indeed, to learn everything we want to know about the flow emanating from the store, we will have to examine the details — the *process specification* — of the process to which the flow is connected; process specifications are discussed in Chapter 11.

There is one procedural detail we can be sure of: the store is passive, and data will not travel from the store along the flow unless a process explicitly asks for them. There is another procedural detail that is assumed, by convention, for information-processing systems: The store is not changed when a packet of information moves from the store along the flow. A programmer might refer to this as a nondestructive read; in other words, a copy of the packet is retrieved from the store, and the store remains in its original condition.[9]

A flow to a store is often described as a write, an update, or possibly a delete. Specifically, it can mean any of the following things:

- One or more new packets are being put into the store. Depending on the nature of the system, the new packets may be *appended* (i.e., somehow arranged so that they are "after" the existing packets); or they may be placed somewhere between existing packets. This is often an implementation issue (i.e., controlled by the specific database management system) in which case the systems analyst ought not to worry about it. It may, however, be a matter of user policy.
- One or more packets are being deleted, or removed, from the store.
- One or more packets are being modified or changed. This may involve a change to *all* of a packet, or (more commonly) just a portion of a packet, or a portion of multiple packets. For example, suppose that a law enforcement agency maintains a store of suspected criminals and that each packet contains the suspect's name and address; the agency might offer a new "identity" to a cooperative suspect, in which case *all* the information pertaining to that suspect's packet would change. As an alternative, consider a **CUSTOMERS** store containing information about customers residing in New York City; if the Post Office decided to change the zip code, or if the telephone company decides to change the area code (both of which have happened to individual neighborhoods within the city over the years), it would necessitate a change to one portion of several packets.

In all these cases, it is evident that the store is changed as a result of the flow entering the store. It is the process (or processes) connected to the other end of the flow that is responsible for making the change to the store.

One point that should be evident from all the examples shown thus far: flows connected to a store can only carry packets of information that the store is capable of holding. Thus, a flow connected to a **CUSTOMERS** store can only carry customer-related information that the store contains; it cannot carry inventory packets or manufacturing packets or astronomical data.

The Terminator

The next component of the DFD is a *terminator*; it is graphically represented as a rectangle, as shown in Figure 9.13. Terminators represent external entities with which the system communicates. Typically, a terminator is a person or a group of people, for example, an outside organization or government agency, or a group or department that is *within* the same company or organization, but *outside* the control of the system being modeled. In some cases, a terminator may be another system, for example, some other computer system with which your system will communicate.

Figure 9.13: Graphical representation of a terminator; source: Image:Figure913.graffle

It is usually very easy to identify the terminators in the system being modeled. Sometimes the terminator is the user; that is, in your discussions with the user, she will say, "I intend to provide data items X, Y, and Z to your system, and I expect the system to provide me with data items A, B, and C." In other cases, the user considers herself part of the system and will help you identify the relevant terminators; for example, she will say to you, "We have to be ready to receive Type 321 forms from the Accounting Department, and we have to send weekly Budget Reports to the Finance Committee." In this last case, it is evident that the Accounting Department and the Finance Committee are separate terminators with which the system communicates.

There are three important things that we must remember about terminators:

1. They are *outside* the system we are modeling; the flows connecting the terminators to various processes (or stores) in our system represent the interface between our system and the outside world.
2. As a consequence, it is evident that neither the systems analyst nor the systems designer are in a position to change the contents of a terminator or the way the terminator works. In the language of several classic textbooks on structured analysis, the terminator is outside the domain of change. What this means is that the systems analyst is modeling a system with the intention of allowing the systems designer a considerable amount of flexibility and freedom to choose the best (or most efficient, or most reliable, etc.) implementation possible. The systems designer may implement the system in a considerably different way than it is currently implemented; the systems analyst may choose to model the requirements of the system in such a way that it looks

considerably different than the way the user mentally imagines the system now (more on this in Section 9.4 and Part III). *But the systems analyst cannot change the contents, or organization, or internal procedures associated with the terminators*.

3. Any relationship that exists *between* terminators will not be shown in the DFD model. There may indeed be several such relationships, but, by definition, those relationships are not part of the system we are studying. Conversely, if there *are* relationships between the terminators, and if it is essential for the systems analyst to model those requirements in order to properly document the requirements of the system, then, by definition, the terminators are actually part of the system and should be modeled as processes.

In the simple examples discussed thus far, we have seen only one or two terminators. In a typical real-world system, there may be literally dozens of different terminators interacting with the system. Identifying the terminators and their interaction with the system is part of the process of building the environmental model, which we will discuss in Chapter 17.

GUIDELINES FOR CONSTRUCTING DFDs

In the preceding section, we saw that dataflow diagrams are composed of four simple components: processes (bubbles), flows, stores, and terminators. Armed with these tools, you can now begin interviewing users and constructing DFD models of systems.

However, there are a number of additional guidelines that you need in order to use DFDs successfully. Some of these guidelines will help you avoid constructing DFDs that are, quite simply, *wrong* (i.e., incomplete or logically inconsistent). And some of the guidelines are intended to help you draw a DFD that will be pleasing to the eye, and therefore more likely to be read carefully by the user.

The guidelines include the following:

1. Choose meaningful names for processes, flows, stores, and terminators.
2. Number the processes.
3. Redraw the DFD as many times as necessary for esthetics.
4. Avoid overly complex DFDs.
5. Make sure the DFD is internally consistent and consistent with any associated DFDs.

Choosing Meaningful Names

As we have already seen, a process in a DFD may represent a *function* that is being carried out, or it may indicate how the function is being carried out, by identifying the person, group, or mechanism involved. In the latter case, it is obviously important to accurately label the process so that the people reading the DFD, especially the users, will be able to confirm that it is an accurate model. However, if the process is carried out by an individual person, I recommend that you identify the role that the person is carrying out, rather than the person's name or identity. Thus, rather than drawing a process like the one shown in Figure 9.14(a), with Fred's name immortalized for all to see, we suggest that you represent the process as shown in Figure 9.14(b). The reason for this is threefold:

- Fred may be replaced next week by Mary or John. Why invite obsolescence in the model?
- Fred may be carrying out several different jobs in the system. Rather than drawing three different bubbles, each labeled Fred but meaning something different, it's better to indicate the actual job that is being done — or at least the role that Fred is playing at the moment (as modeled in each of their bubbles).

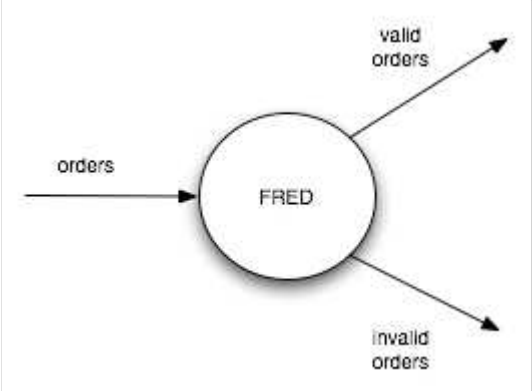


Figure 9.14(a): An inappropriate process name;
source: Image:Figure914a.graffle

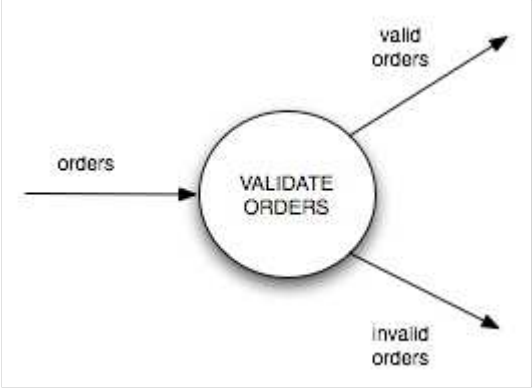


Figure 9.14(b): A more appropriate process name;
source: Image:Figure914b.graffle

- Identifying Fred is likely to draw attention to the way Fred happens to carry out the job at hand. As we will see in Part III, we will generally want to concentrate on the underlying *business policy* that must be carried out, without reference to the procedures (which may be based on customs and history no longer relevant) used to carry out that policy.

If we are lucky enough to avoid names of people (or groups) and political roles altogether, we can label the processes in such a way as to identify the *functions* that the system is carrying out. A good discipline to use for process names is a verb and an object. That is, choose an active verb (a transitive verb, one that takes an object) and an appropriate object to form a descriptive phrase for your process. Examples of process names are:

- CALCULATE MISSILE TRAJECTORY
- PRODUCE INVENTORY REPORT
- VALIDATE PHONE NUMBER
- ASSIGN STUDENTS TO CLASS

You will find, in carrying out this guideline, that it is considerably easier to use specific verbs and objects if the process itself is relatively simple and well defined. Even in the simple cases, though, there is a temptation to use wishy-washy names like DO, HANDLE, and PROCESS. When such “elastic” verbs are used (verbs whose meaning can be stretched to cover almost any situation), it often means that the systems analyst is not sure what function is being performed or that several functions have been grouped together but don’t really belong together. Here are some examples of poor process names:

- DO STUFF
- MISCELLANEOUS FUNCTIONS
- NON-MISCELLANEOUS FUNCTIONS
- HANDLE INPUT
- TAKE CARE OF CUSTOMERS
- PROCESS DATA
- GENERAL EDIT

The names chosen for the process names (as well as flow names and terminator names) should come from a vocabulary that is meaningful to the user. This will happen quite naturally if the DFD is drawn as a result of a series of interviews with the users *and* if the systems analyst has some minimal understanding of the underlying subject matter of the application. But two cautions must be kept in mind:

1. There is a natural tendency for users to use the specific abbreviation and acronyms that they are familiar with; this is true for both the processes and the flows that they describe. Unfortunately, this usually results in a DFD that is very heavily oriented to the way things happen to be done now. Thus, the user might say, "Well, we get a copy of Form 107 — it's the pink copy, you know — and we send it over to Joe where it gets frogulated." A good way to avoid such excessively idiosyncratic terms is to choose verbs (like "frogulate") and objects (like "Form 107") that would be meaningful to someone in the same industry or application, but working in a different company or organization. If you're building a banking system, the process names and flow names should, ideally, be understandable to someone in a different bank.
2. If the DFD is being drawn by someone with a programming background, there will be a tendency to use such programming-oriented terminology as "ROUTINE," "PROCEDURE," "SUBSYSTEM," and "FUNCTION," even though such terms may be utterly meaningless in the user's world. Unless you hear the users using these words in their own conversation, avoid them in your DFD.

Number the Processes

As a convenient way of referencing the processes in a DFD, most systems analysts number each bubble. It doesn't matter very much how you go about doing this — left to right, top to bottom, or any other convenient pattern will do -- *as long as you are consistent in how you apply the numbers*.

The only thing that you must keep in mind is that the numbering scheme will imply, to some casual readers of your DFD, a certain *sequence* of execution. That is, when you show the DFD to a user, he may ask, "Oh, does this mean that bubble 1 is performed first, and then bubble 2, and then bubble 3?" Indeed, you may get the same question from other systems analysts and programmers; anyone who is familiar with a flowchart may make the mistake of assuming that numbers attached to bubbles imply a sequence.

This is not the case at all. The DFD model is a network of communicating, asynchronous processes, which is, in fact, an accurate representation of the way most systems actually operate. Some sequence may be implied by the presence or absence of data (e.g., it may turn out that bubble 2 cannot carry out its function until it receives data from bubble 1), but the numbering scheme has nothing to do with this.

So why do we number the bubbles at all? Partly, as indicated above, as a convenient way of referring to the processes; it's much easier in a lively discussion about a DFD to say "bubble 1" rather than "EDIT TRANSACTION AND REPORT ERRORS." But more importantly, the numbers become the basis for a *hierarchical* numbering scheme when we introduce leveled dataflow diagrams in Section 9.3.

Avoid Overly Complex DFDs

The purpose of a DFD is to accurately model the functions that a system has to carry out and the interactions between those functions. But another purpose of the DFD is to be read and understood, not only by the systems analyst who constructed the model, but by the users who are the experts in the subject matter. This means that the DFD should be

readily understood, easily absorbed, and pleasing to the eye.

We will discuss a number of esthetic guidelines in the next subsection, but there is one overriding guideline to keep in mind: *don't create a DFD with too many processes, flows, stores, and terminators*. In most cases, this means that you should't have more than half a dozen processes and related stores, flows, and terminators on a single diagram.[10] Another way of saying this is that the DFD should fit comfortably onto a standard 8.5- by 11-inch sheet of paper.

There is one major exception to this, as we will discuss in Chapter 18: a special DFD known as a *context diagram'* that represents an entire system as a single process and highlights the interfaces between the system and the outside terminators. Figure 9.15 shows a typical context diagram, and you can see that it is enough to scare away many systems analysts, not to mention the unwary user! Typically, context diagrams like the one shown in Figure 9.15 cannot be simplified, for they are depicting, even at the highest level of detail, a reality that is intrinsically complex.[11]

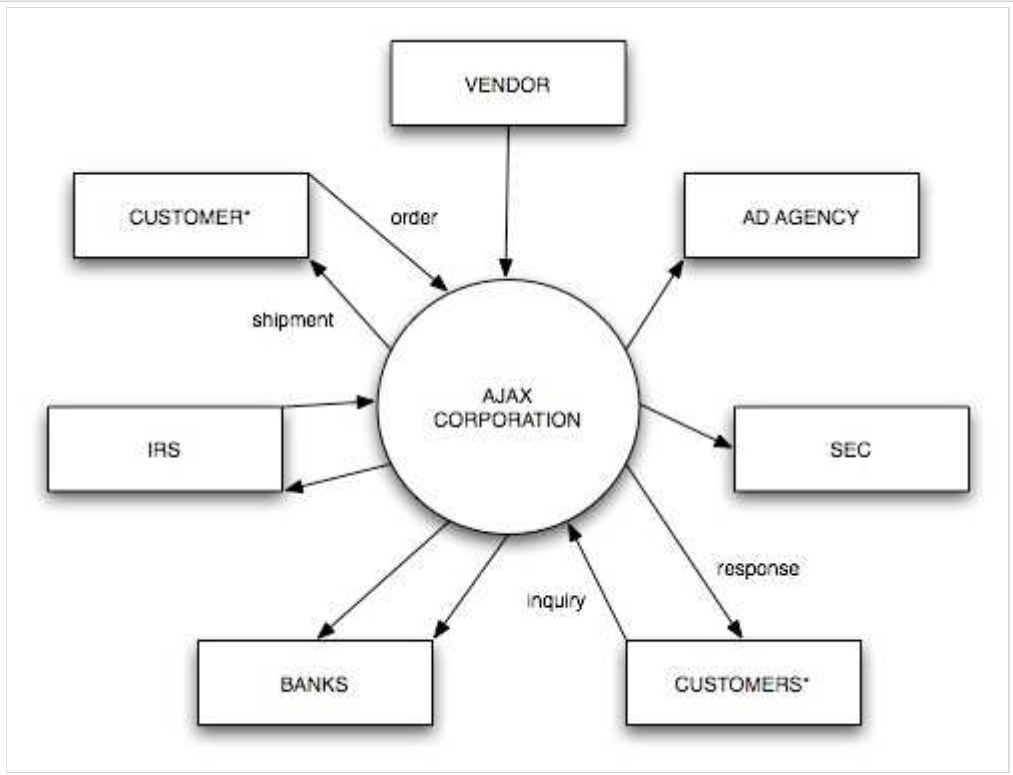


Figure 9.15: A typical context diagram; source: Image:Figure915.graffle

Redraw the DFD As Many Times As Necessary

In a real-world systems analysis project, the DFD that we have discussed in this chapter will have to be drawn, redrawn, and redrawn again, often as many as ten times or more, before it is (1) technically correct, (2) acceptable to the user, and (3) neatly enough drawn that you wouldn't be embarrassed to show it to the board of directors in your organization. This may seem like a lot of work, but it is well worth the effort to develop an accurate, consistent, esthetically pleasing model of the requirements of your system. The same is true of any other engineering discipline: would you want to fly in an airplane designed by engineers who got bored with their engineering drawings after the second iteration?[12]

What makes a dataflow diagram esthetically pleasing? This is obviously a matter of personal taste and opinion, and it may be determined by standards set by your organization or by the idiosyncratic features of any automated workstation-based diagramming package that you use. And the user's opinion may be somewhat different from yours; within reason, whatever the user finds esthetically pleasing should determine the way you draw your diagram. Some of the issues that will typically come up for discussion in this area are the following:

- *Size and shape of bubbles*. Some organizations draw dataflow diagrams with rectangles or ovals instead of circles; this is obviously a matter of esthetics. Also, some users become quite upset if the bubbles in the DFD are not all the same size: they think that if one bubble is bigger than another, it means that that part of the system is more important or is different in some other significant way. (In fact, it usually happens only because the bubble's name was so long that the systems analyst had to draw a larger bubble just to encompass the name!)
- *Curved dataflows versus straight dataflows*. To illustrate this issue, consider the DFDs in Figure 9.16(a) and (b). Which one is more esthetically pleasing? Many observers will shrug their shoulders and say, "They're really both the same." But others — and this is the point! — will choose one and violently reject the other. It's obviously a good idea to know in advance which choice will be accepted and which will be rejected. In roughly the same category is the issue of crossed arrows; are they allowed or not allowed?
- *Hand-drawn diagrams versus machine-generated diagrams*. There is hardly any excuse for hand-drawn diagrams today, since the typical systems analyst is almost certain to have access to a PC with reasonable drawing tools.[13] The issue here, though, is the user's reaction to these diagrams: some have a strong preference for the machine-generated diagrams because they're "neater," while others prefer handdrawn pictures because it gives them the feeling that the diagram hasn't been finalized or "frozen" yet, and that they can still make changes.

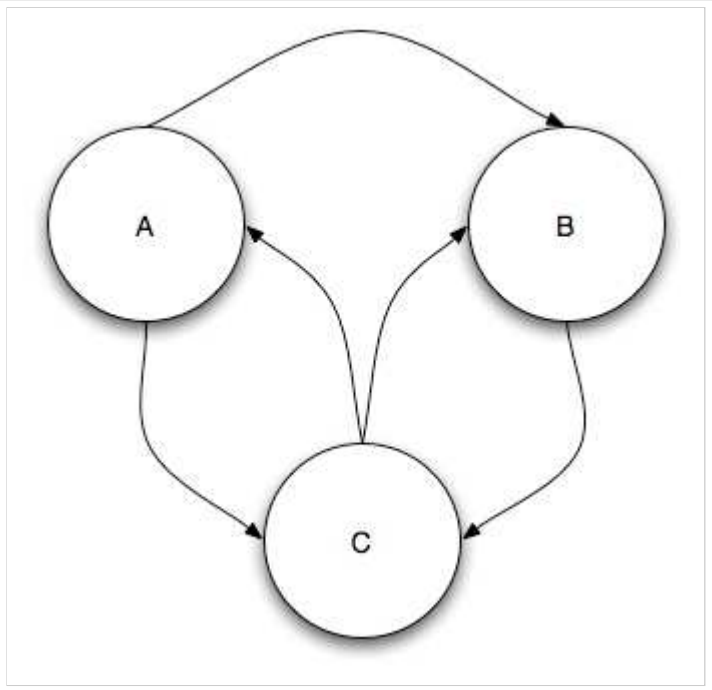


Figure 9.16(a): One version of a DFD; source: Image:Figure916a.graffle

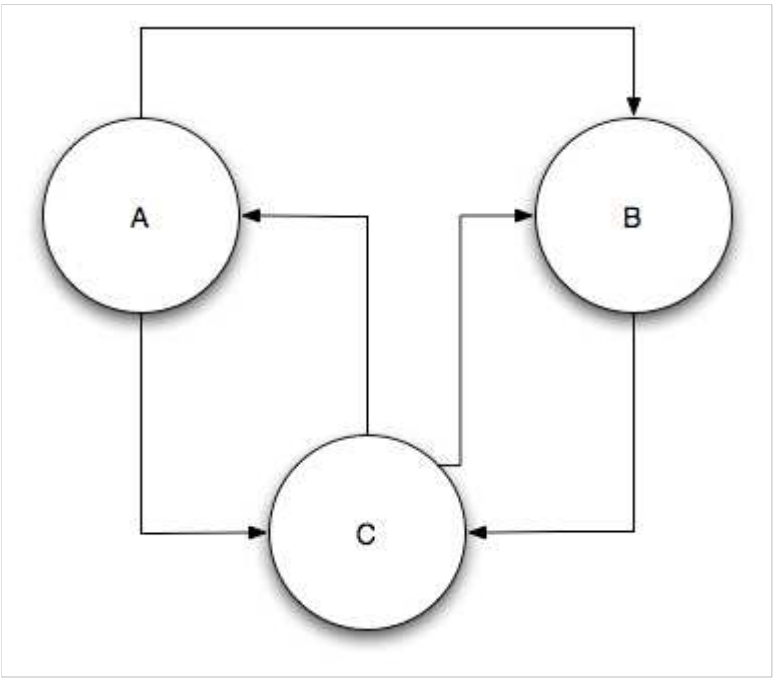


Figure 9.16(b): A different version of the DFD; source: Image:Figure916b.graffle

Make Sure That Your DFD Is Logically Consistent

As we will see in Chapter 14, a number of rules and guidelines that help ensure the dataflow diagram is consistent with the other system models -- the entity-relationship diagram, the state-transition diagram, the data dictionary, and the process specification. However, there are some guidelines that we use now to ensure that the DFD itself is consistent. The major consistency guidelines are these:

- *Avoid infinite sinks*, bubbles that have inputs but no outputs. These are also known by systems analysts as "black holes," in an analogy to stars whose gravitational field is so strong that not even light can escape. An example of an infinite sink is shown in Figure 9.17.
- *Avoid spontaneous generation bubbles*; bubbles that have outputs but no inputs are suspicious, and generally incorrect. One plausible example of an output-only bubble is a random-number generator, but it is hard to imagine any other reasonable example. A typical output-only bubble is shown in Figure 9.18.

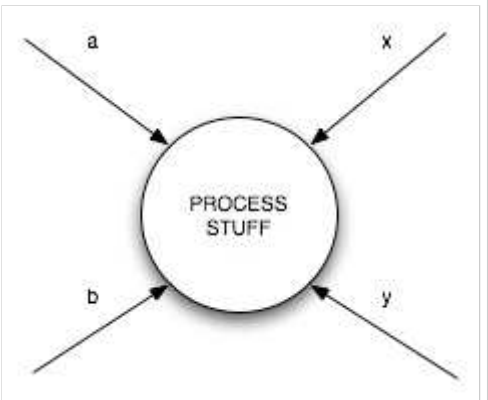


Figure 9.17: An example of an infinite sink; source: Image:Figure917.graffle

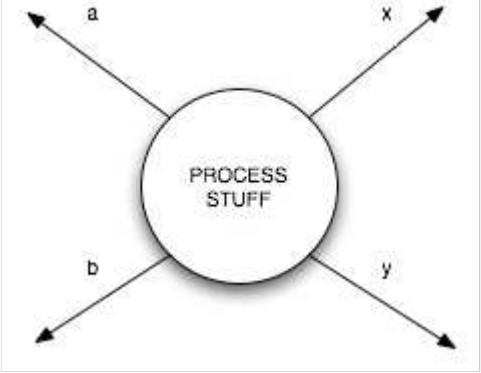


Figure 9.18: An example of an output-only bubble; source: Image:Figure918.graffle

- *Beware of unlabeled flows and unlabeled processes*. This is usually an indication of sloppiness, but it may mask an even deeper error: sometimes the systems analyst neglects to label a flow or a process because he or she simply cannot think of a reasonable name. In the case of an unlabeled flow, it may mean that several unrelated elementary data items have been arbitrarily packaged together; in the case of an unlabeled process, it may mean that the systems analyst was so confused that he or she drew a disguised flowchart instead of a dataflow diagram.[14]
- *Beware of read-only or write-only stores*. This guideline is analogous to the guideline about input-only and output-only processes; a typical store should have both inputs *and* outputs.[15] The only exception to this guideline is the external store, a store that serves as an interface between the system and some external terminator. Figure 9.19 shows an example of a stock market system with a legitimate write-only store.

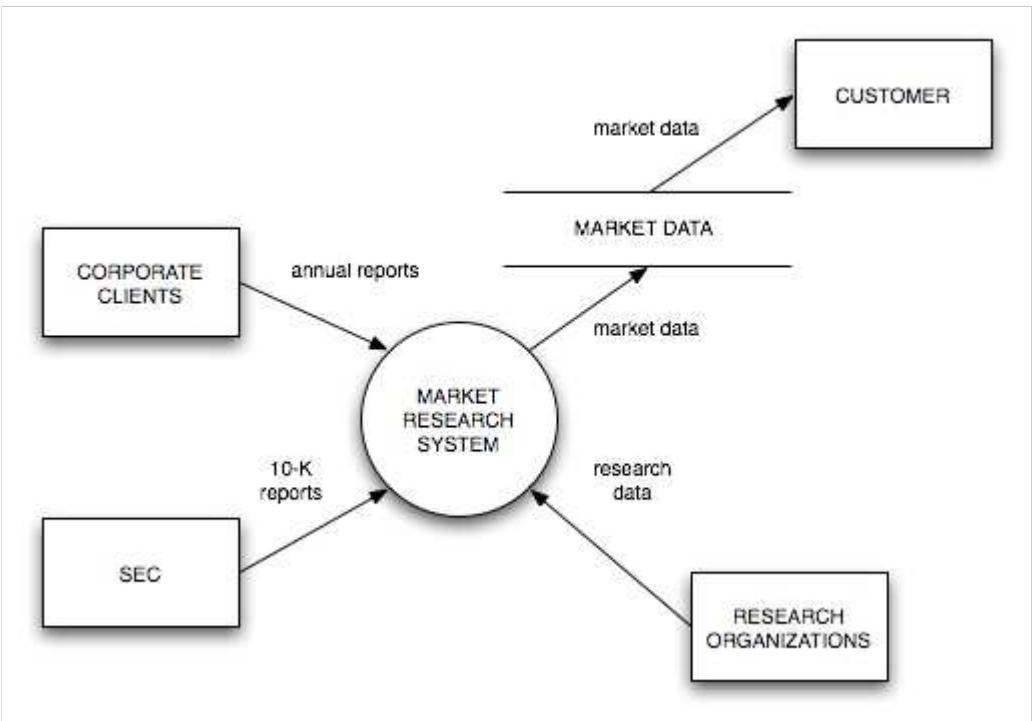


Figure 9.19: A legitimate case of a write-only store; source: Image:Figure919.graffle

LEVELED DFDs

Thus far in this chapter, the only complete dataflow diagrams we have seen are the simple three-bubble system shown in Figure 9.1 and the one-bubble system shown in Figure 9.19. But DFDs that we will see on *real* projects are considerably larger and more complex. Consider, for example, the DFD shown in Figure 9.20. Can you imagine showing this to a typical user?

Section 9.2.3 already suggested that we should avoid diagrams such as the one depicted in Figure 9.20. But how? If the system is intrinsically complex and has dozens or even hundreds of functions to model, how can we avoid the kind of DFD shown in Figure 9.20?

The answer is to organize the overall DFD in a series of levels so that each level provides successively more detail about a portion of the level above it. This is analogous to the organization of maps in an atlas, as we discussed in Chapter 8: we would expect to see an overview map that shows us an entire country, or perhaps even the entire world; subsequent maps would show us the details of individual countries, individual states within countries, and so on. In the case of DFDs, the organization of levels is shown conceptually in Figure 9.21.

The top-level DFD consists of only one bubble, representing the entire system; the dataflows show the interfaces between the system and the external terminators (together with any external stores that may be present, as illustrated by Figure 9.19). This special DFD is known as a *context diagram* and is discussed in Chapter 18.

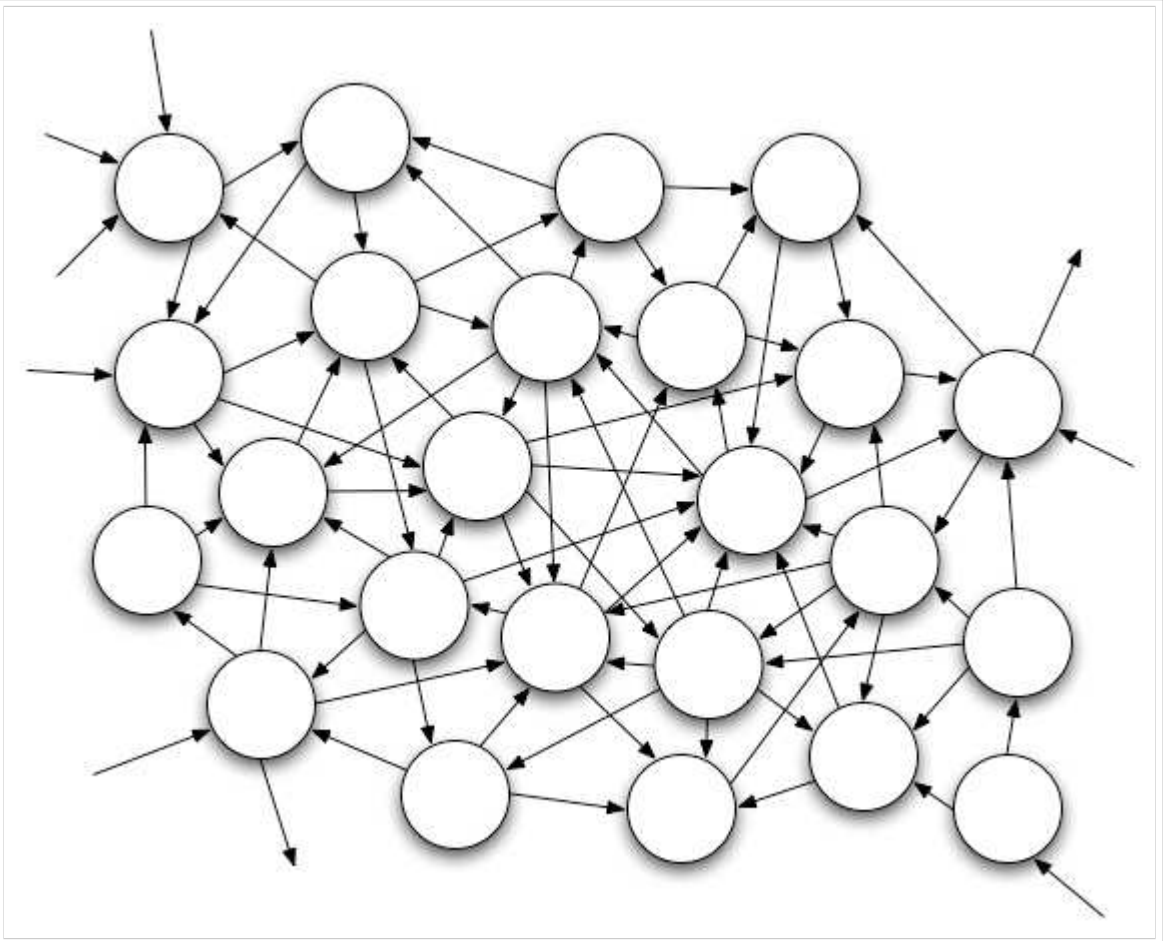


Figure 9.20: A complex DFD; source: Image:Figure920.graffle

The DFD immediately beneath the context diagram is known as Figure 0. It represents the highest-level view of the major functions within the system, as well as the major interfaces between those functions. As discussed in Section 9.2.2, each of these bubbles should be numbered for convenient reference.

The numbers also serve as a convenient way of relating a bubble to the next lower level DFD which more fully describes that bubble. For example:

- Bubble 2 in Figure 0 is associated with a lower-level DFD known as Figure 2. The bubbles within Figure 2 are numbered 2.1, 2.2, 2.3, and so on.
- Bubble 3 in Figure 0 is associated with a lower-level DFD known as Figure 3. The bubbles within Figure 3 are numbered 3.1, 3.2, 3.3, and so on.
- Bubble 2.2 in Figure 2 is associated with a lower-level DFD known as Figure 2.2. The bubbles within Figure 2.2 are numbered 2.2.1, 2.2.2, 2.2.3, and so on.
- If a bubble has a name (which indeed it should have!), then that name is carried down to the next lower level figure. Thus, if bubble 2.2 is named **COMPUTE SALES TAX**, then Figure 2.2, which partitions bubble 2.2 into more detail, should be labeled **"Figure 2.2: COMPUTE SALES TAX."**

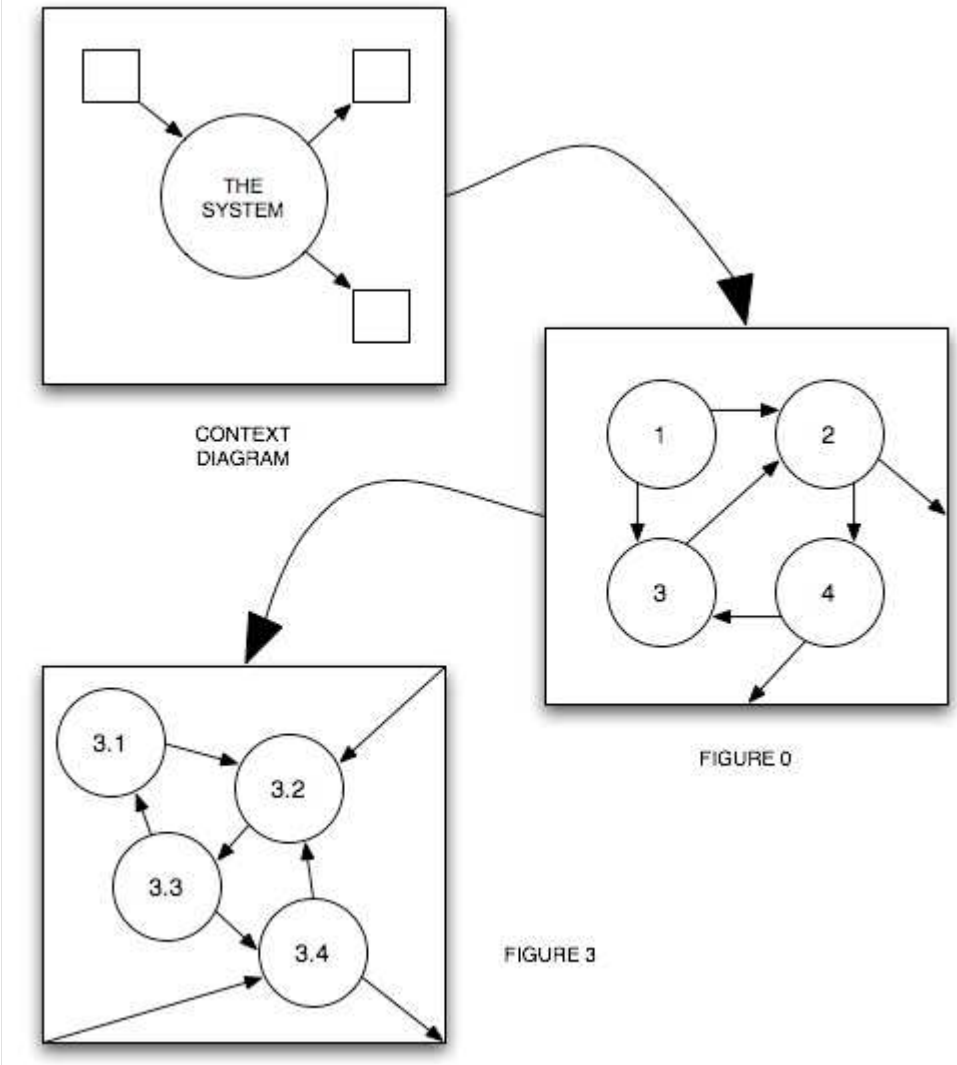


Figure 9.21: Leveled dataflow diagrams; source: Image:Figure921.graffle

As you can see, this is a fairly straightforward way of organizing a potentially enormous dataflow diagram into a group of manageable pieces. But there are several things we must add to this description of leveling:

1. *How do you know how many levels you should have in a DFD?* The answer to this was suggested in Section 9.2.3: each DFD figure should have no more than half a dozen bubbles and related stores. Thus, if you have partitioned a large system into three levels, but your bottom-level DFDs still contain 50 bubbles each, you have at least one more level to go. An alternative checkpoint is provided in Chapter 11, when we discuss process specifications for the bubbles in the bottom-level DFDs: if we can't write a reasonable process specification for a bubble in about one page, then it probably is too complex, and it should be partitioned into a lower level DFD

before we try to write the process specification.

2. *Are there any guidelines about the number of levels one should expect in a typical system?* In a simple system, one would probably find two to three levels; a medium-size system will typically have three to six levels; and a large system will have five to eight levels. You should be extremely wary of anyone who tells you that all systems can be modeled in exactly three levels; such a person will also try to sell you the Brooklyn Bridge. On the other hand, remember that the total number of bubbles increases exponentially as we go from one level to the next lower level. If, for example, each figure has seven bubbles, then there will be 343 bubbles at the third level, 16,807 bubbles at the fifth level, and 40,353,607 bubbles at the ninth level.
3. *Must all parts of the system be partitioned to the same level of detail?* For example, if bubble 2 in Figure 0 requires three more levels of detail, is it necessary for bubble 3 to also have three more levels of detail: that is, a Figure 3; a set of figures labeled Figure 3.1, Figure 3.2, ...; and a set of figures labeled 3.1.1, 3.1.2, ..., 3.2.1, 3.2.2, and so on. The answer is "No." Some parts of a system may be more complex than others and may require one or two more levels of partitioning. On the other hand, if bubble 2, which exists in the top-level Figure 0, turns out to be primitive, while bubble 3 requires seven levels of further partitioning, then the overall system model is skewed and has probably been poorly organized. In this case, some portions of bubble 3 should be split out into a separate bubble or perhaps reassigned to bubble 2.
4. *How do you show these levels to the user?* Many users will only want to look at *one* diagram: a high-level executive user may only want to look at the context diagram or perhaps at Figure 0; an operational-level user may only want to look at the figure that describes the area of the system in which she or he is interested. But if someone wants to look at a large part of the system or perhaps the entire system, then it makes sense to present the diagrams in a top-down fashion: begin with the context diagram and work your way down to lower levels of detail. It is often handy to have two diagrams side by side on the desk (or displayed with an overhead projector) when you do this: the diagram that you are particularly interested in looking at, plus the parent diagram that provides a high-level context.
5. *How do you ensure that the levels of DFDs are consistent with each other?* The issue of consistency turns out to be critically important, because the various levels of DFDs are typically developed by different people in a real-world project; a senior systems analyst may concentrate on the context diagram and Figure 0, while several junior systems analysts work on Figure 1, Figure 2, and so on. To ensure that each figure is consistent with its higher-level figure, we follow a simple rule: *the dataflows coming into and going out of a bubble at one level must correspond to the dataflows coming into and going out of an entire figure at the next lower level which describes that bubble.* Figure 9.22(a) shows an example of a balanced dataflow diagram; Figure 9.22(b) shows two levels of a DFD that are out of balance.
6. *How do you show stores at the various levels?* This is one area where redundancy is deliberately introduced into the model. The guideline is as follows: show a store at the highest level where it first serves as an interface between two or more bubbles; then show it again in EVERY lower-level diagram that further describes (or partitions) those interface bubbles. Thus, Figure 9.23 shows a store that is shared by two high-level processes, **A** and **B**; the store would be shown again on the lower-level figures that further describe **A** and **B**. The corollary to this is that local stores, which are used only by bubbles in a lower-level figure, will not be shown at the higher levels, as they will be subsumed into a process at the next higher level.
7. *How do you actually DO the leveling of DFDs?* The discussion thus far has been misleading in a rather subtle way: while it is true that the DFDs should be presented to a user audience in a top-down fashion, it is not necessarily true that the systems analyst should develop the DFDs in a top-down fashion. The top-down approach is intuitively very appealing: one can imagine beginning with the context diagram, and then developing Figure 0, and then methodically working down to the lower levels of detail.[16] However, we will see in Chapter 17 that there are problems with this approach; a more successful approach is to first identify the external *events* to which the system must respond, and to use those events to create a rough, "first-cut" DFD. In Chapter 20, we will see that this first-cut DFD may have to be partitioned *upward* (to create higher-level DFDs) and *downward* (to create lower-level DFDs). For now, it is sufficient that you simply realize that the organization and presentation of a leveled set of DFDs does not necessarily correspond to the strategy for developing those levels in the first place.

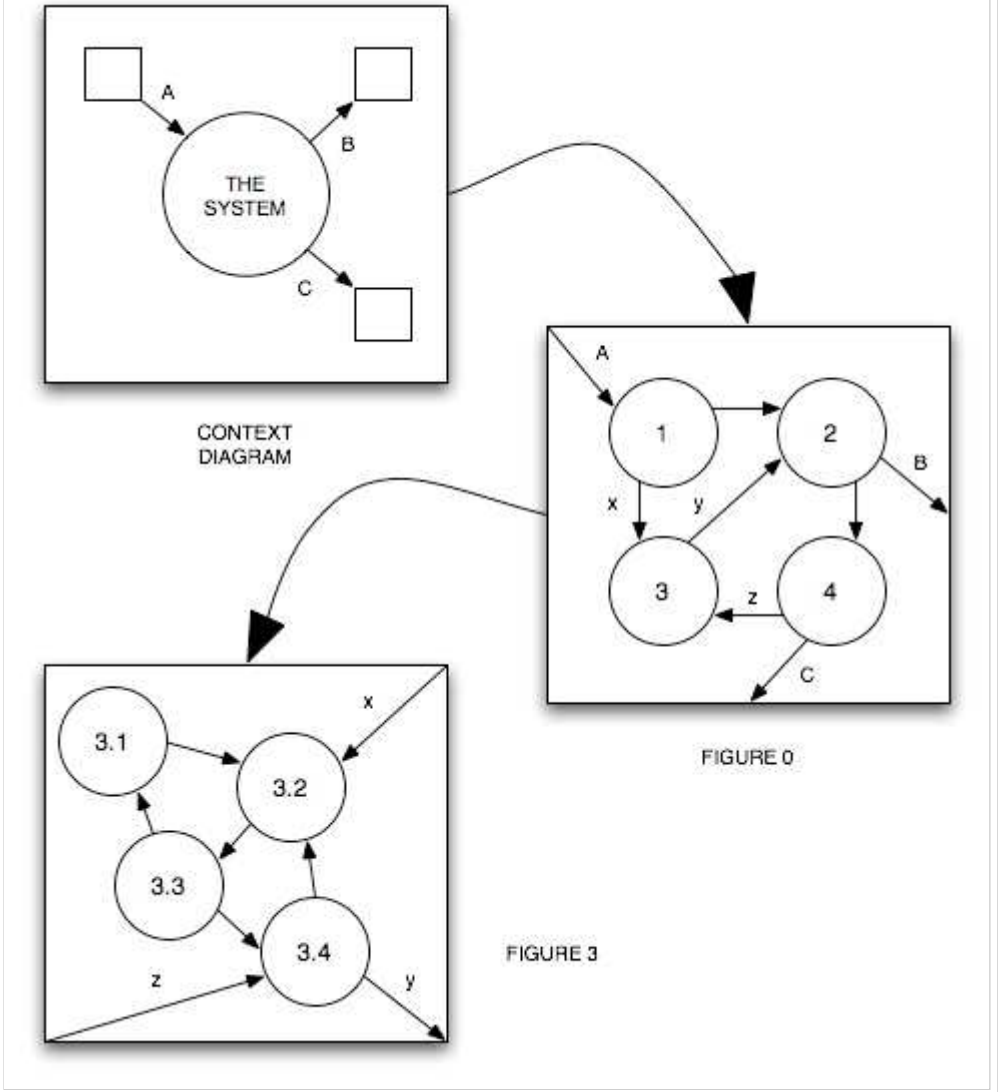


Figure 9.22(a): A balanced DFD fragment; source: Image:Figure922a.graffle

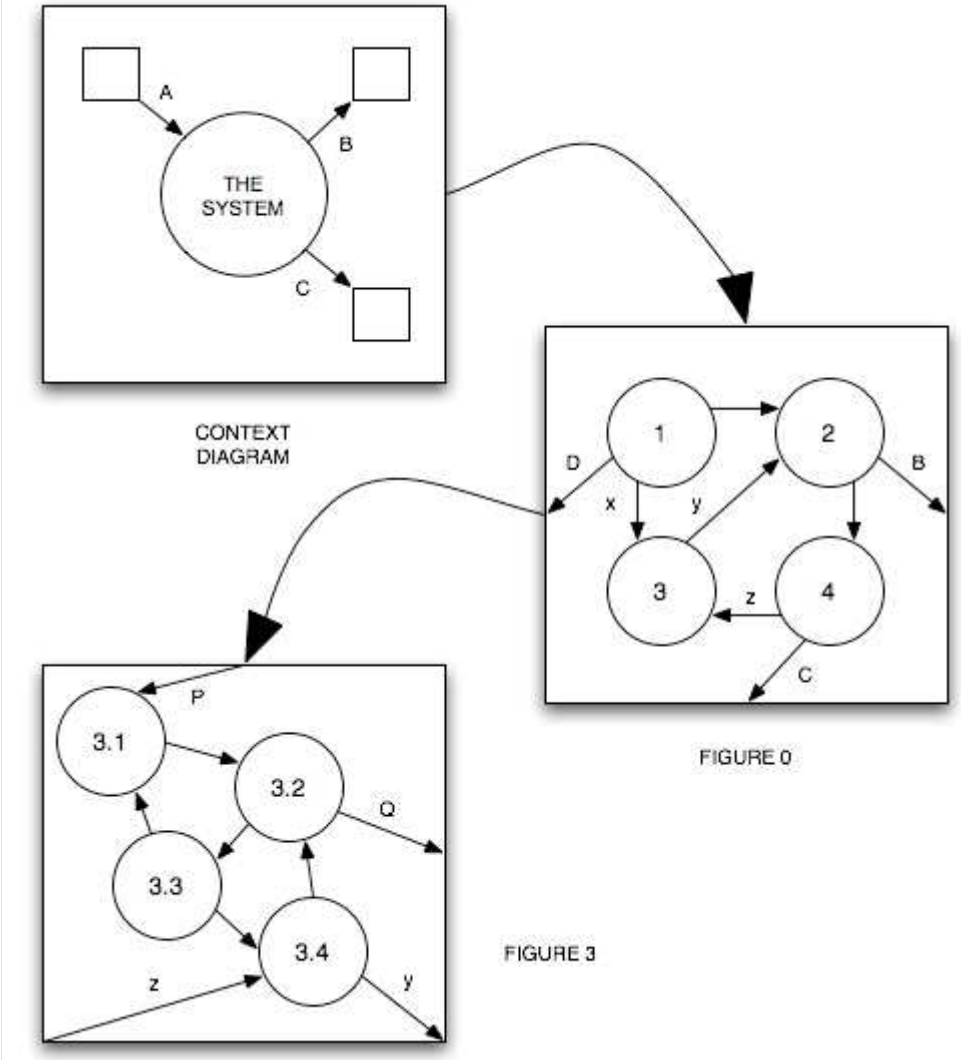


Figure 9.22(b): A DFD fragment that is out of balance; source: Image:Figure922b.graffle

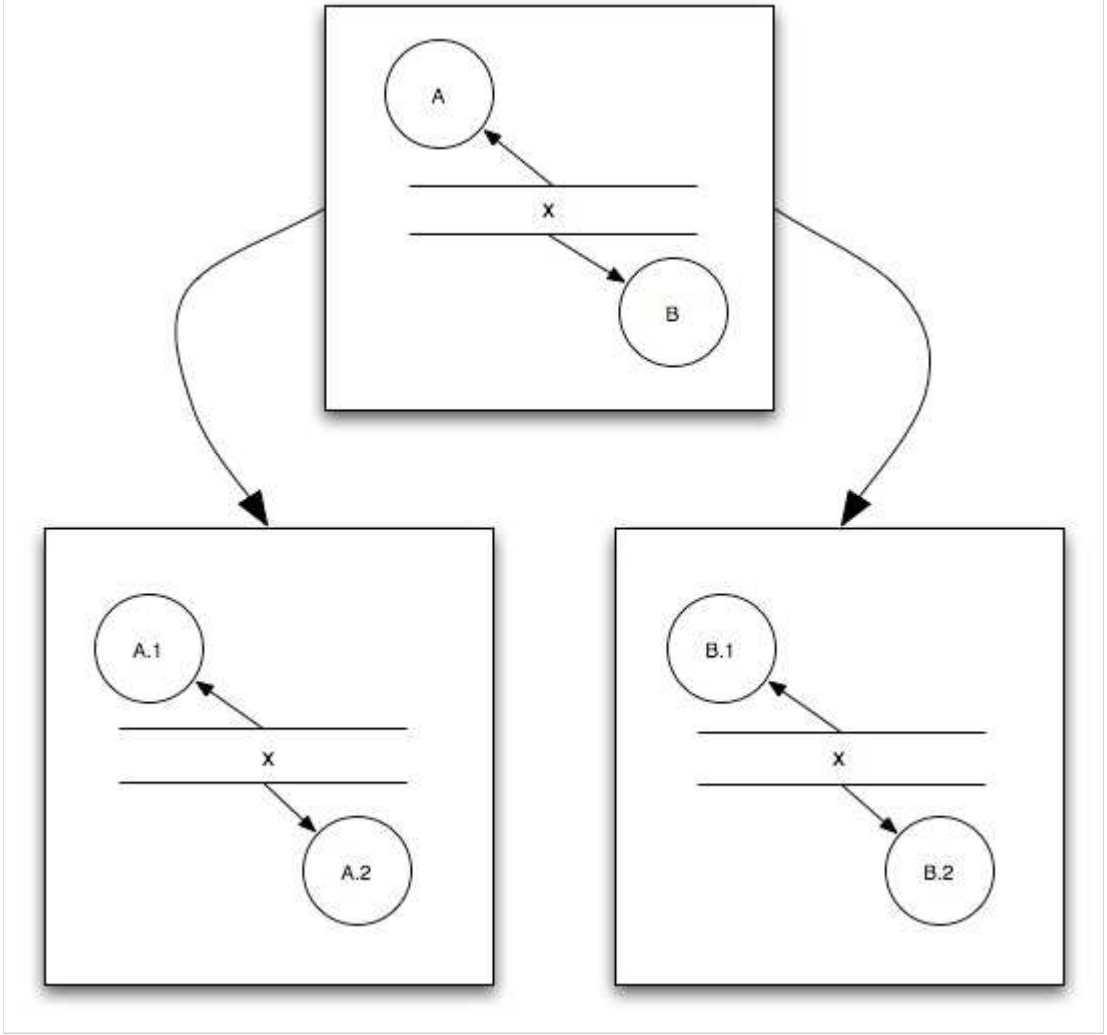


Figure 9.23: Showing stores at lower levels; source: Image:Figure923.graffle

Extensions to the DFD for real-time systems

The flows discussed throughout this chapter are data flows; they are pipelines along which packets of data travel between processes and stores. Similarly, the bubbles in the DFDs we have seen up to now could be considered processors of data. For a very large class of systems, particularly business systems, these are the only kind of flows that we need in our system model. But for another class of systems, the *real-time* systems, we need a way of modeling *control flows* (i.e., signals or interrupts). And we need a way to show *control processes* — (i.e., bubbles whose only job is to coordinate and synchronize the activities of other bubbles in the DFD).[17] These are shown graphically with dashed lines on the DFD, as illustrated in Figure 9.24.

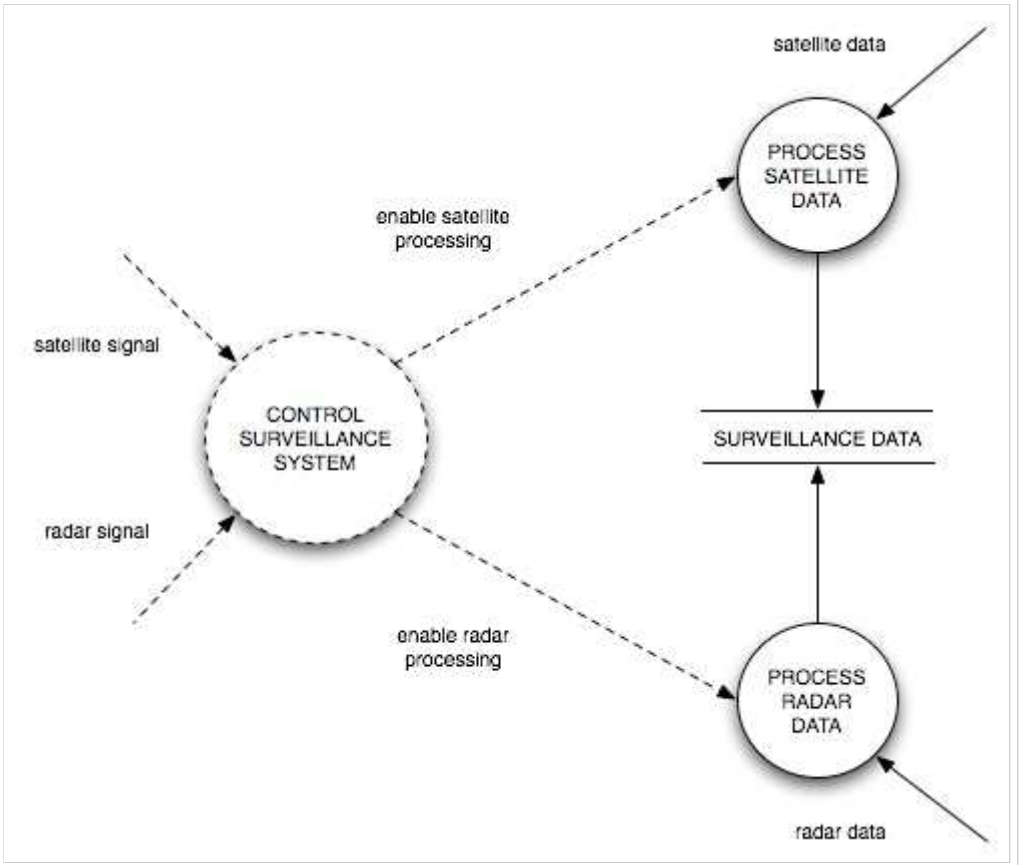


Figure 9.24: A DFD with control flows and control processes; source: Image:Figure924.graffle

A control flow may be thought of as a pipeline that can carry a binary signal (i.e., it is either on or off). Unlike the other flows discussed in this chapter, the control flow does not carry value-bearing data. The control flow is sent from one process to another (or from some external terminator to a process) as a way of saying, "Wake up! It's time to do your job." The implication, of course, is that the process has been dormant, or idle, prior to the arrival of the control flow.

A control *process* may be thought of as a supervisor or executive bubble whose job is to coordinate the activities of the other bubbles in the diagram; its inputs and outputs consist *only* of control flows. The outgoing control flows from the control process are used to wake up other bubbles; the incoming control flows generally indicate that one of the bubbles has finished carrying out some task, or that some extraordinary situation has arisen, which the control bubble needs to be informed about. There is typically only one such control process in a single DFD.

As indicated above, a control flow is used to wake up a normal process; once awakened, the normal process proceeds to carry out its job as described by a process specification (see Chapter 11). The internal behavior of a control process is different, though: this is where the time-dependent behavior of the system is modeled in detail. The inside of a control process is modeled with a *state-transition diagram*, which shows the various states that the entire system can be in and the circumstances that lead to a change of state. State-transition diagrams are discussed in Chapter 13.

SUMMARY

As we have seen in this chapter, the dataflow diagram is a simple but powerful tool for modeling the functions in a system. The material in Sections 9.1, 9.2, and 9.3 should be sufficient for modeling most classical business-oriented information systems. If you are working on a real-time system (e.g., process control, missile guidance, or telephone switching), the real-time extensions discussed in Section 9.4 will be important; for more detail on real-time issues, consult (Ward and Mellor, 1985).

Unfortunately, many systems analysts think that dataflow diagrams are all they need to know about structured analysis.

If you ask one of your colleagues if he is familiar with structured analysis, he is likely to remark, "Oh, yeah, I learned about all those bubbles and stuff." On the one hand, this is a tribute to the power of dataflow diagrams — it is often the only thing that a systems analyst remembers after reading a book or taking a course on structured analysis! On the other hand, it is a dangerous situation: without the additional modeling tools presented in the following chapters, the dataflow diagrams are worthless. Even if the data relationships and time-dependent behavior of the system are trivial (which is unlikely), it is still necessary to combine DFDs with the data dictionary (discussed in Chapter 10) and the process specification (discussed in Chapter 11).

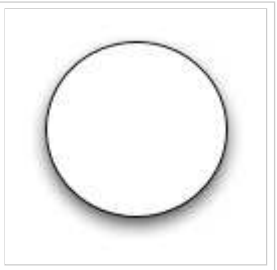
So don't put the book down yet! There's more to learn!

REFERENCES

1. Wayne Stevens, Glen Myers, and Larry Constantine, "Structured Design." (http://www.research.ibm.com/journal/sj/132/ibmsj1302C.pdf) *IBM Systems Journal* (http://www.research.ibm.com/journal/sj45-4.html) , May 1974.
2. Ed Yourdon and Larry Constantine, *Structured Design* (http://www.amazon.com/exec/obidos/ASIN/0138544719/edyourdonswebsit) . New York: YOURDON Press, 1975.
3. Glen Myers, *Reliable Software through Composite Design* (http://www.amazon.com/exec/obidos/ASIN/0442256205/edyourdonswebsit) . New York: Petrocelli/Charter, 1975.
4. Tom DeMarco, *Structured Analysis and Systems Specification* (http://www.amazon.com/exec/obidos/ASIN/0138543801/edyourdonswebsit) . Englewood Cliffs, NJ: Prentice-Hall, 1979.
5. Chris Gane and Trish Sarson, *Structured Systems Analysis: Tools and Techniques* (http://www.amazon.com/exec/obidos/ASIN/0930196007/edyourdonswebsit) . Englewood Cliffs, NJ: Prentice-Hall, 1978.
6. Doug Ross and Ken Schoman, Jr., "Structured Analysis for Requirements Definition." (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=35896&arnumber=1702398&count=17&index=3) *IEEE Transactions on Software Engineering* (http://www.computer.org/tse/) , January 1977, pp. 41-48. Also reprinted in Ed Yourdon, *Classics in Software Engineering* (http://www.amazon.com/exec/obidos/ASIN/0917072146/edyourdonswebsit) . New York: YOURDON Press, 1979.
7. Paul Ward and Steve Mellor, *Structured Development of Real-Time Systems* (http://www.amazon.com/exec/obidos/ASIN/013854803X/edyourdonswebsit) , Volumes 1-3. New York: YOURDON Press, 1986.
8. Edsger W. Dijkstra, "Cooperating Sequential Processes." *Programming Languages*, F. Genuys (editor). New York: Academic Press, 1968.
9. Paul Ward, "The Transformation Schema: An Extension of the Dataflow Diagram to Represent Control and Timing." *IEEE Transactions on Software Engineering* (http://www.computer.org/tse/) , February 1986, pp.198-210.
10. Derek Hatley, "The Use of Structured Methods in the Development of Large Software-Based Avionics Systems." *AIAA/IEEE 6th Digital Avionics Conference*, Baltimore, 1984.
11. M. Webb and Paul Ward, "Executable Dataflow Diagrams: An Experimental Implementation." *Structured Development Forum*, Seattle, August 1986.
12. E. Reilly and J. Brackett, "An Experimental System for Executing Real-Time Structured Analysis Models." *Proceedings of the 12th Structured Methods Conference*, Chicago, August 1987.
13. J. Robertson and S. Robertson, *Complete Systems Analysis: The Workbook, the Textbook, the Answers* (http://www.amazon.com/exec/obidos/ASIN/0932633501/edyourdonswebsit) . New York: Dorset House, 1998.

QUESTIONS AND EXERCISES

1. Give a brief description of a dataflow diagram. What is the difference between a DFD and a flowchart?
2. List six synonyms for dataflow diagram.
3. What can DFDs be used for besides modeling information systems?
4. What are the four major components of a DFD?
5. What are three common synonyms for process in a DFD?
6. Is there any significance to the choice of a circle for a process? Would it be better to use a triangle or a hexagon? Why or why not?
7. What is wrong with the following process?



Source: Image:Quest7.graffle

8. What is wrong with the following process?

Source: Image:Quest8.graffle

9. What is wrong with the following process?

Source: Image:Quest9.graffle

10. What is wrong with the following process?

Source: Image:Quest10.graffle

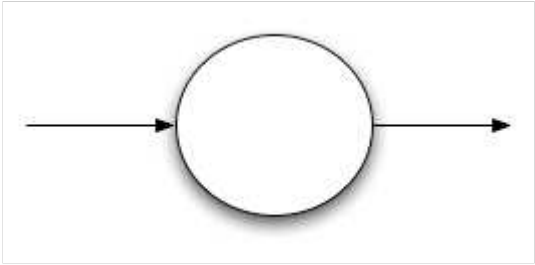
11. What is wrong with the following process?

Source: Image:Quest11.graffle

12. Why would a systems analyst draw a DFD with a process consisting of the name of a person or organizational

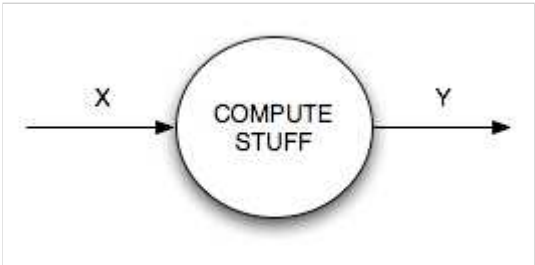
group?

13. Are flows on a DFD restricted to just showing the movement of information? Could they show the movement of anything else?
14. What is wrong with this DFD?



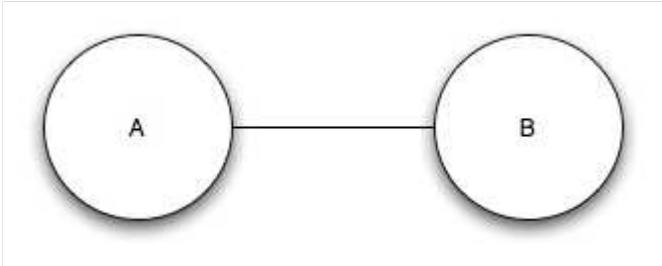
Source: Image:Quest14.graffle

15. How can the same chunk of data have different meanings in a DFD? Draw an example of such a situation.
16. What is the significance of the following DFD?



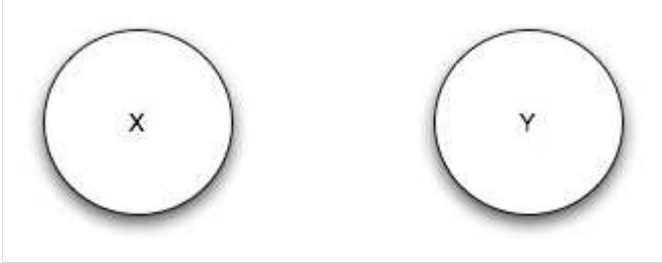
Source: Image:Quest16.graffle

17. Is there any limit to the number of inputs and outputs that a process can have in a DFD? What would your reaction be if you saw a process with 100 inputs and 100 outputs?
18. What is wrong with the following DFD?



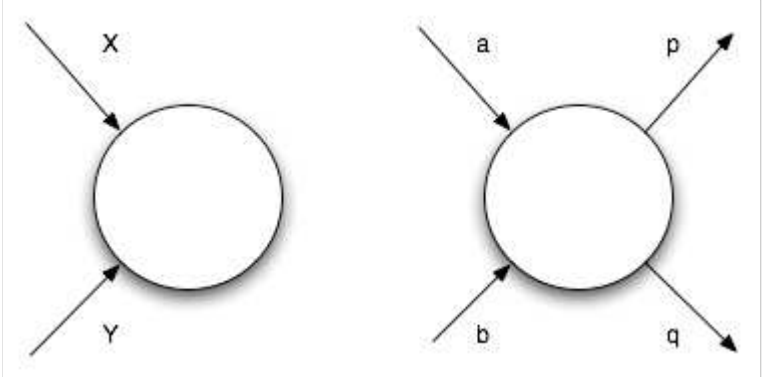
Source: Image:Quest18.graffle

19. What is wrong with the following DFD?



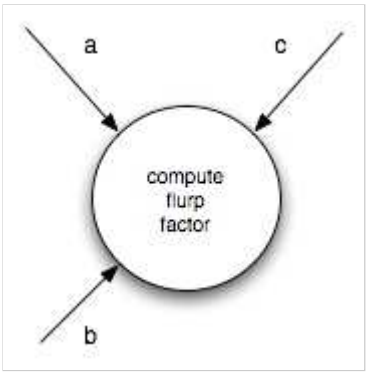
Source: Image:Quest19.graffle

20. What is wrong with the following DFDs?



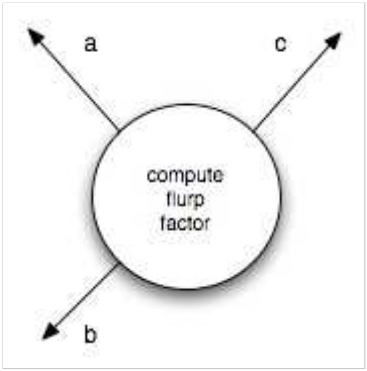
Source: Image:Quest20.graffle

21. What is wrong with the following DFD?



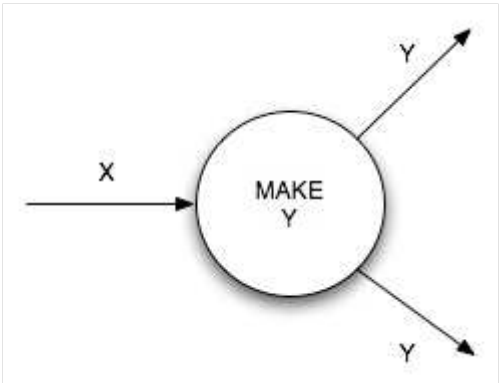
Source: Image:Quest21.graffle

22. What is wrong with the following DFD?



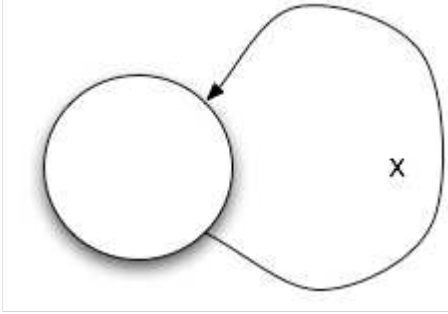
Source: Image:Quest22.graffle

23. Give a description of a dialogue flow.
24. Is the following DFD valid? Is there any other way to draw it?



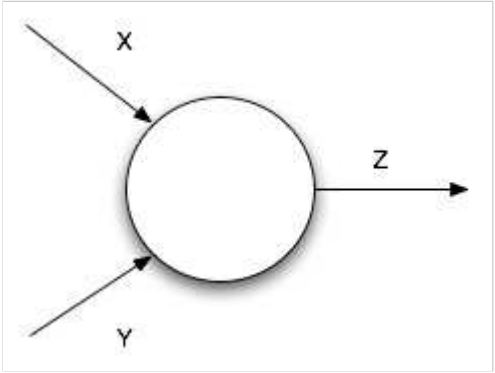
Source: Image:Quest24.graffle

25. Is the following DFD valid? Is there any other way to draw it?



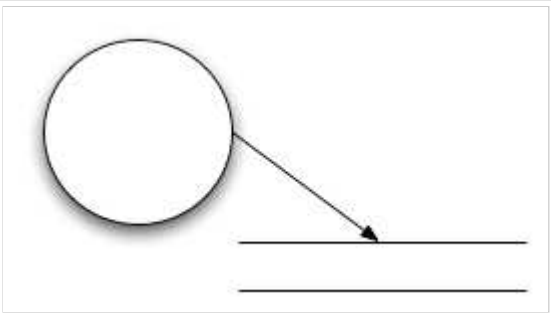
Source: Image:Quest25.graffle

26. Under what circumstances would you expect to see duplicate copies of an output flow from a process?
27. Why do DFDs avoid showing procedural details?
28. In the diagram below, how many x and how many y elements are required to produce one z output?



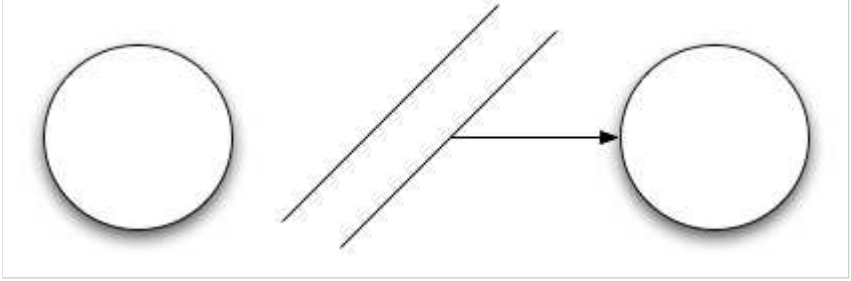
Source: Image:Quest28.graffle

29. What does a store show in a DFD?
30. What is the convention for naming stores in a DFD?
31. What are alternative names for a store? Is it acceptable to use the term file? Why or why not?
32. What names are commonly used to describe packets of information in a store?
33. What are the four common reasons for showing implementation stores in a DFD?
34. Do you think implementation stores should be allowed in a DFD? Why or why not?
35. What is the meaning of an unlabeled flow into or out of a store?
36. Is there any limit to the number of flows into or out of a store? If so, state what the limit is.
37. What is wrong with the following DFDs?
    - (a)



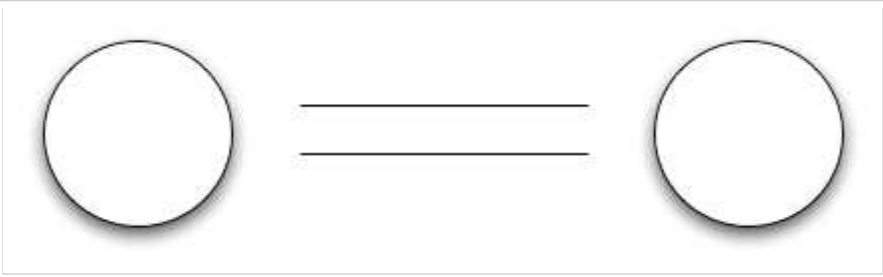
Source: Image:Quest37a.graffle

    - (b)



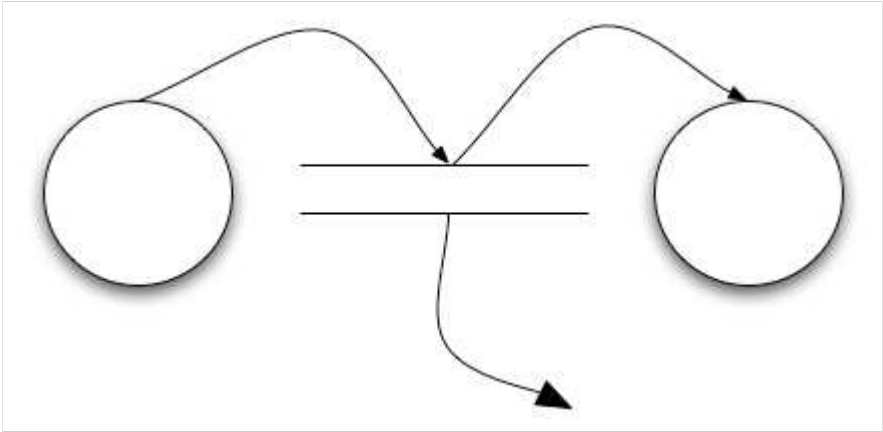
Source: Image:Quest37b.graffle

- (c)



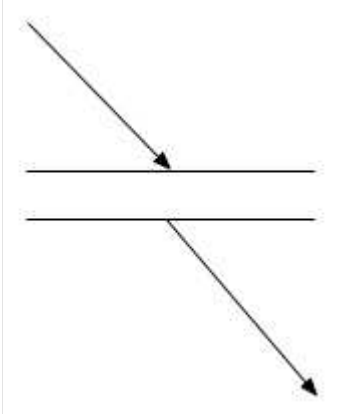
Source: Image:Quest37c.graffle

- (d)



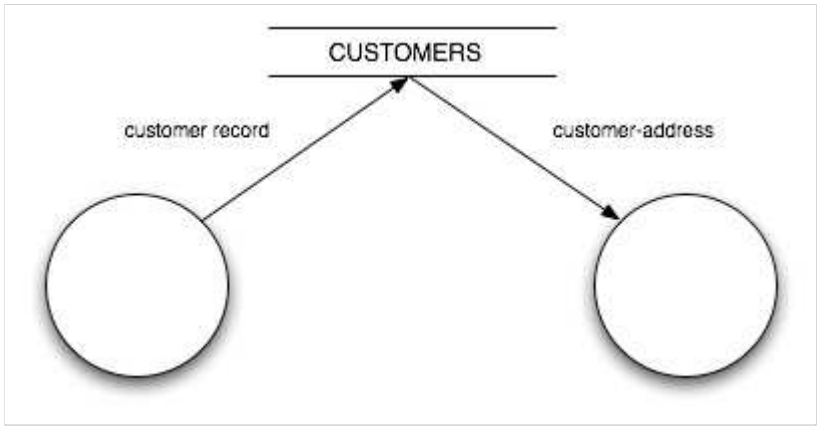
Source: Image:Quest37d.graffle

- (e)



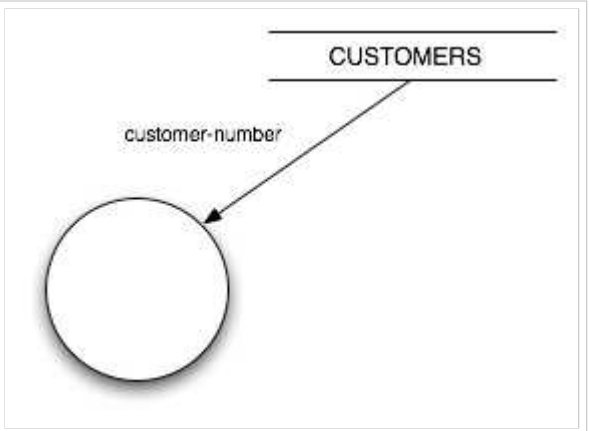
Source: Image:Quest37e.graffle

38. What are the four possible interpretations of a dataflow from a store into a process?
39. Does the following DFD make sense? Why or why not?



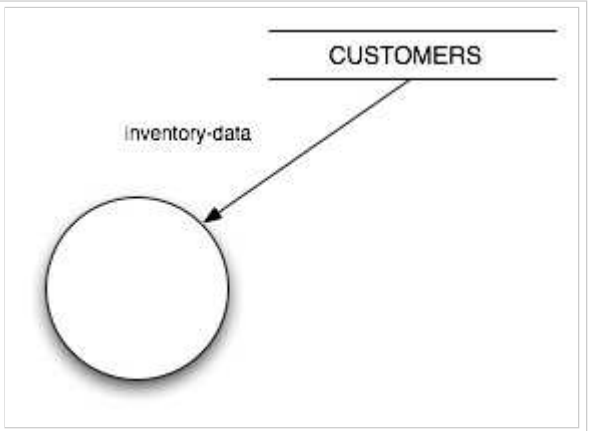
Source: Image:Quest39.graffle

40. Give an example of a situation where a process might extract portions of more than one record from a store in a single logical access.
41. Give an example of a situation where a process might extract more than one packet of information from a store in a single logical access.
42. Can you tell from looking only at the diagrams whether the following DFDs are correct?
    - (a)



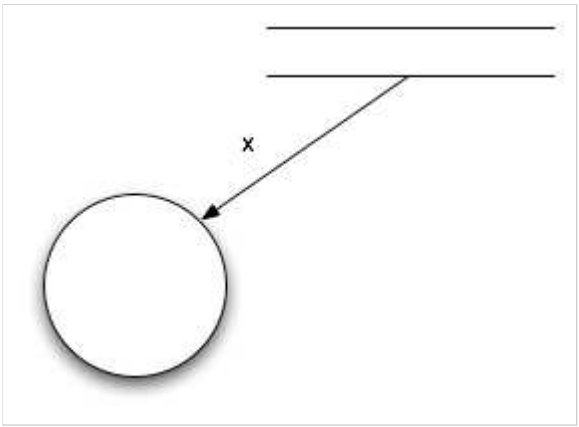
Source: Image:Quest42a.graffle

    - (b)



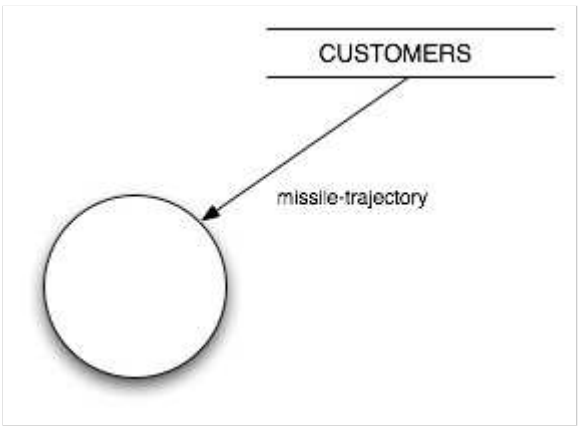
Source: Image:Quest42b.graffle

    - (c)



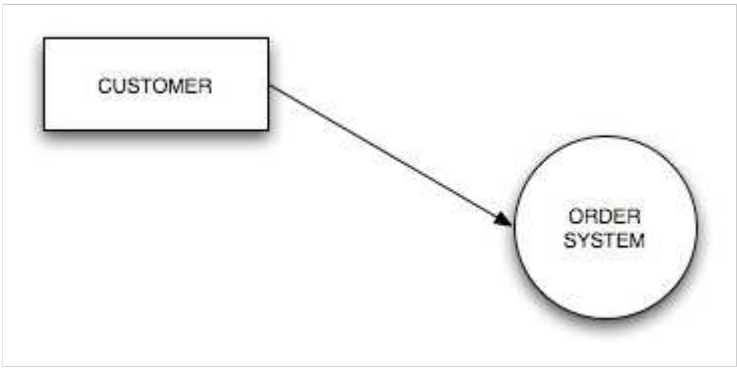
Source: Image:Quest42c.graffle

43. What happens to a store after data have moved from the store, along a flow, to a process? Is this true of all types of systems or just information systems?
44. What are the three possible interpretations of a flow into a store?
45. How do we show packets of data being deleted from a store?
46. What is wrong with the following DFD?



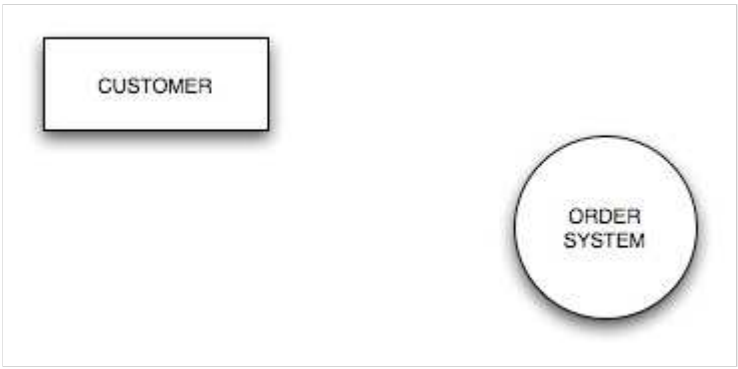
Source: Image:Quest46.graffle

47. What is the purpose of showing a terminator on a DFD?
48. How should the systems analyst identify the terminators?
49. What do the flows between terminators and processes represent?
50. What is wrong with the following DFDs?
    - (a)



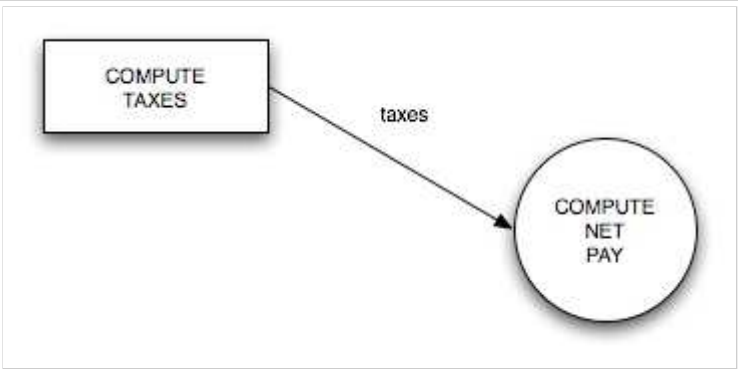
Source: Image:Quest50a.graffle

    - (b)



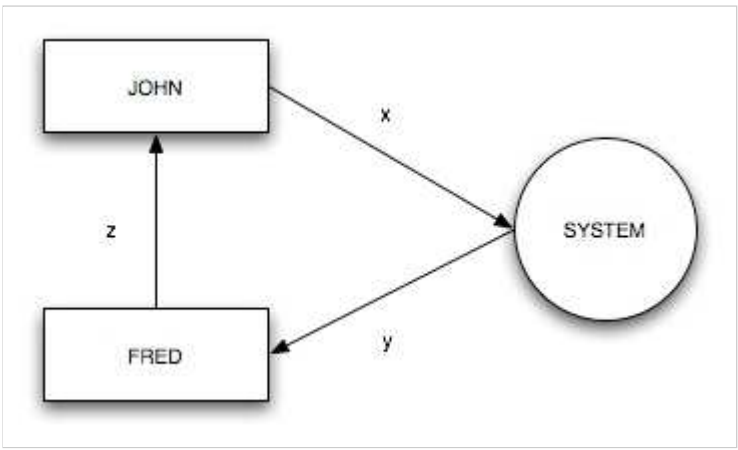
Source: Image:Quest50b.graffle

- (c)



Source: Image:Quest50c.graffle

- (d)



Source: Image:Quest50d.graffle

51. Is the systems analyst allowed to change the contents or organization of a terminator as part of her project? What if she feels strongly that it should be changed?
52. Why should a process not show the name of the person currently performing that function?
53. What is a good guideline for naming the processes in a DFD?
54. Give five examples of process names that you would not like to see in a DFD.
55. How can you tell whether the name chosen for a process is likely to be meaningful?
56. What is the misinterpretation that a user is likely to make of bubble numbers in a DFD?
57. How can you tell whether a DFD is likely to be too complex for the user to comprehend?
58. How rigidly enforced should the complexity guideline be? Should any exceptions be allowed? Why or why not?
59. Why is it often necessary to redraw a DFD many times?
60. What are the major issues that determine whether a DFD is esthetically pleasing? Do you think that these issues can be expressed as standards?
61. Do your colleagues prefer bubbles or ovals for processes? Which do you prefer?
62. Do you think the dataflows between processes should be shown as curved lines or straight lines? Can you think of any advantage or disadvantage of either approach? Is it an important issue?
63. What is an infinite sink? Why should it be considered an error in a typical DFD?

64. What are spontaneous generation bubbles in a DFD? Why should they be avoided in a typical DFD?
65. Why are unlabeled flows and processes dangerous?
66. Why are read-only stores and write-only stores typically an error in a DFD?
67. Why are leveled DFDs important in a system model?
68. How many levels of DFD should the systems analyst expect to see in a typical large system? Can you suggest an upper limit on the number of levels in a DFD?
69. In a leveled DFD:
    - (a) What would we call the "children" bubbles below bubble 2.3?
    - (b) What would be the figure name of the figure in which bubble 2.3 appears?
    - (c) How many higher-level figures are there above the figure in which bubble 2.3 appears? What are they called?
70. Is it necessary for all parts of a system to be partitioned to the same level of detail? Why or why not?
71. Suppose someone showed you a DFD in which bubble 1 was partitioned into 2 lower levels, and bubble 2 was partitioned into 17 lower levels. What would your reaction be? What recommendations, if any, would you make?
72. What does balancing mean, in the context of this chapter? How can you tell whether a DFD is balanced?
73. Why is Figure 9.22(b) in this chapter out of balance?
74. What is the guideline for showing stores at different levels of a DFD?
75. What is a local store? What are the guidelines for showing a local store in a leveled DFD?
76. *Research Project*: What is the relationship between the guideline for showing local stores and the concept of object-oriented design? To pursue this further, read Chapters 19 and 20.
77. What problems might be associated with developing a set of leveled DFDs in a top-down fashion (as compared to reading an already developed set of DFDs in a top-down fashion)?
78. What is a control flow? Why is it different from a dataflow?
79. What is a control process? Why is it different from a normal process in a DFD?
80. What is a control store? Why is it different from a normal store in a DFD?
81. Draw a dataflow diagram to model the following recipe for Fruits de Mer au Riz (mixed sfellfish with rice), taken from *The New York Times 60-Minute Gourmet*, by Pierre Franey (New York: TIMES Books, 1979):
    - To begin, prepare and measure out all the ingredients for the rice. To save time, chop an extra cup of onions and 2 extra cloves of garlic for the seafood mixture. Set these aside.
    - Heat the 2 tablespoons of oil for the rice in a saucepan and add 1/4 cup of onion, green pepper, and 1 clove of garlic and cook until wilted. Add the saffron and cook about 2 minutes longer.
    - Add the rice, water, salt, and pepper and cover closely. Cook about 17 minutes or just until the rice is tender. As the rice cooks, prepare the seafood. Remember that when the rice is cooked, remove it from the heat. It can stand several minutes, covered, without damage.
    - In a kettle, heat the 1/4 cup oil and add the 1 cup of onions and 2 cloves of garlic. Stir and cook until wilted. Add the red pepper, tomatoes, wine, and oregano. Add salt and pepper. Cover and cook for 10 minutes.
    - Add the clams and mussels to the kettle and cover again. Cook about 3 minutes and add the shrimp, scallops, salt and pepper to taste. Cover and cook about 5 minutes.
82. Draw a dataflow diagram for the following recipe for Coquille St. Jacques Meuniere (scallops quick-fried in butter), taken from *The New York Times 60-Minute Gourmet*, by Pierre Franey (New York: TIMES Books, 1979):
    - A point to be made a hundred times is organization. Before you cook, chop what has to be chopped and measure what has to be measured. Bring out such pots and pans as are necessary for cooking — in this case, two skillets (one for the scallops, one for the tomatoes) and a saucepan (for the potatoes).
    - Empty the scallops into a bowl and add the milk, stirring to coat. Let stand briefly.
    - Place the flour in a dish and add salt and pepper to taste. Blend well. Drain the scallops. Dredge them in flour and add them to a large sieve. Shake to remove excess flour. Scatter them onto a sheet of foil or wax paper so that they do not touch or they might stick together.
    - The scallops must be cooked over high heat without crowding. Heat 3 tablespoons of the oil and 1 tablespoon of butter in a large skillet. When the mixture is quite hot but not smoking, add half of the scallops, shaking and tossing them in the skillet so that they cook quickly and evenly until golden brown on all sides.
    - Use a slotted spoon and transfer the scallops to a hot platter. Add remaining 2 tablespoons of oil to the skillet and when it is quite hot, add the remaining scallops, shaking and tossing them in the skillet as before. When brown, transfer them to the platter with the other scallops. Wipe out the skillet, add the remaining butter and cook until lightly browned or the color of hazelnuts. Sprinkle over the scallops. Then sprinkle scallops with the lemon juice and chopped parsley.

83. Draw a dataflow diagram for the following recipe for Omelette Pavillon (omelet with chicken, tomato, and cheese), taken from *The New York Times 60-Minute Gourmet*, by Pierre Franey (New York: TIMES Books, 1979):
    - Just before you start cooking the omelets, break three eggs for each omelet into as many individual bowls as needed for the number of omelets to be made. Add salt and pepper to taste and about two teaspoons of heavy cream. You may also beat the eggs to expediate making the omelets.
    - Heat 2 tablespoons off butter in a saucepan and add the flour. Stir with a wire whisk until blended. Add the chicken broth and cook, stirring vigorously with the whisk. Add the cream and bring to a boil. Simmer for about 10 minutes.
    - Meanwhile, heat another tablespoon of butter in a saucepan and add the onion. Cook, stirring, until wilted, and add the tomatoes, thyme, bay leaf, salt and pepper. Simmer, stirring occasionally, about 10 minutes.
    - Heat another tablespoon of butter and add the chicken. Cook, stirring about 30 seconds. Add 3 tablespoons of the cream sauce. Bring to the boil and remove from the heat. Set aside.
    - To the remaining cream sauce add the egg yolk and stir to blend. Add salt and pepper to taste and the grated Swiss cheese. Heat, stirring, just until the cheese melts. Set aside.
    - Beat the eggs with salt and pepper. Add 6 tablespoons of the tomato sauce. Heat the remaining 3 tablespoons of butter in an omelet pan or a Teflon skillet, and when it is hot, add the eggs. Cook, sitrring, until the omelet is set on the bottom but moist and runny in the center. Spoon creamed chicken down on the center of the omelet and add the remaining tomato sauce. Quickly turn the omelet out into a baking dish.
    - Spoon the remaining cream sauce over the omelet and sprinkle with grated Parmesan cheese. Run the dish under the broiler until golden brown.

ENDNOTES

1. ↑ However, the disadvantage of MacDraw (and other generic drawing programs like it) is that it does not know anything about the special nature of dataflow diagrams or other system models presented in this book. It knows only about primitive shapes such as rectangles, circles, and lines. Intermediate-level programs like Visio (for Microsoft Windows) and Top-Down (on the Macintosh) are better, because they tend to have sophisticated "templates" of useful diagramming notations, and they also understand that some diagram elements are "connected" (e.g., a flow is connected to a process), so that if one connected element is moved to some other part of the diagram, the other connected element(s) are moved too. The CASE toolsproducts discussed in Appendix A are far more powerful, because they know a great deal about the subject matter of dataflow diagrams, and the rules that distinguish legal from illegal diagrams.
2. ↑ The shape that the systems analyst uses for the process is often associated with a particular "camp" of structured analysis. The circle is associated with the "Yourdon/DeMarco" camp, since it is used in various textbooks published by YOURDON Press. Similarly, the oval shape is often associated with the "Gane/Sarson" camp, since it was introduced by Chris Gane and Trish Sarson in their book (Gane and Sarson, 1977), and was used by consulting organizations such as McDonnell Douglas Automation Company (McAuto) and various other organizations. The rectangle shape is typically associated with the "SADT" camp, since it was popularized in various articles about Softech's Structured Analysis Design Technique (SADT); see, for example, (Ross and Schoman, 1977).
3. ↑ An acceptable alternative to the dialogue is to use two different flows, one showing the input (inquiry) to the process, and the other flow showing the output (response). This is, in fact, a better way to handle things if one input could conceivably lead to several entirely different actions (and responses) from the process. In the case of a simple inquiry-response situation, the use of one dialogue flow or separate input and output flows is a matter of choice. Most systems analysts prefer the dialogue notation because (1) it draws the reader's attention to the fact that the input and output are related to each other, and (2) it reduces the clutter on a DFD with several flows and processes and presents the reader with a cleaner, simpler diagram.
4. ↑ Exactly how this duplication of data packets and/or decomposition of data packets takes place is considered an issue of implementation, that is, something that the systems designer will have to worry about, but not something that the systems analyst needs to show in the model of the system. It may eventually be accomplished by hardware or software, or manually, or by black magic. If the systems analyst is modeling a system that already exists, there may be some temptation to show the mechanism (i.e., the process) that carries out the duplication/decomposition of data. We will discuss this in more detail in Part III.
5. ↑ The notation D1 in Figure 9.9(b) is simply a numbering scheme so that we can distinguish this store from

other stores on the diagram. The convention followed in this book does not involve labeling or numbering the stores (simply because it hasn't seemed necessary or even useful), but (as we will see in Section 9.2), it does involve numbering the bubbles.

6. ↑ It is also common to refer to one packet of information in the store as a record, and to refer to components of each packet as fields. There is nothing wrong with this terminology, but it is used so often in a computer-database context that it is likely to create the same kind of problems discussed above. For the time being, we will use the term packet to describe a single instance of a collection of related objects in the store.
7. ↑ We will mention several such conventions in this chapter, as well as similar conventions concerning the other modeling tools. Your project manager, your organization's standards manual, or the CASE tool that you use for your project (see Appendix A) may force you to use one convention or another; but you should see that there is a certain amount of flexibility to the modeling tools and modeling notation presented here. The important thing is consistency: all the packet-bearing flows into or out of a store should either be consistently labeled or consistently unlabeled.
8. ↑ How do we know that the labels on the flow have anything to do with the components of a packet of information in the store? How do we know, for example, that a flow labeled **PHONE-NUMBER** has anything to do with packets of information in the **CUSTOMERS** store? There is a temptation, especially in a real-world project where everyone is relatively familiar with the subject matter, to simply say, "Oh, that's intuitively obvious! Of course the phone number is a component of a customer packet." But to be sure, we need to see a rigorous definition of the composition of a **CUSTOMERS** packet. This is found in the data dictionary, which we will discuss in Chapter 10.
9. ↑ If you are using a DFD to model something other than a pure information processing system, this may not be true. For example, the store might contain physical items, and the flow might be a mechanism for conveying materials from the store to the process. In this case, a packet would be physically removed from the store, and the store would be depleted as a result. In a system model containing information stores and physical stores, it is important to annotate the DFD (or provide an explanation in the data dictionary) so that the reader will not be confused.
10. ↑ This guideline comes from "The Magical Number Seven, Plus or Minus Two," by George Miller, *Psychology Review*, 1956.
11. ↑ Actually, there are a few things that we can do: if there are several different dataflows between a terminator and the single system bubble, they can be consolidated into a single dataflow. The data dictionary, discussed in Chapter 10, will be used to explain the composition and meaning of the aggregate dataflow. And if the context diagram is being shown to several disparate audiences (e.g., different user groups with different interests), different context diagrams can be drawn to highlight only those terminators and flows that a particular user group in interested in. But, in most cases, there is no escaping the bottom line: if the overall system is intrinsically complex, the context diagram will be, too. More on this in Chapter 18.
12. ↑ Lest you think that airplanes are different from automated information systems or that they are more critical, keep in mind that computer systems now control most of the planes that you fly on; a typical large passenger airplane may have a hundred or more complex computer systems, which in turn interface with such complex real-time systems as the one used by the Federal Aviation Administration to monitor the airspace around airports.
13. ↑ However, as we've already noted in this chapter, a drawing tool like Visio on Microsoft Windows, or Canvas on the Macintosh, is not the same as the so-called CASE tools that can also check the diagram for logical consistency. We'll discuss this issue in more detail in Appendix A.
14. ↑ There is one idiomatic convention that violates this guideline, which we discussed in section 9.1.3: an unlabeled flow into or out of a store is, by convention, an indication that one entire instance (or record) is being put into or taken out of the store.
15. ↑ Sometimes it may not be immediately evident whether the store has both inputs and outputs. As we will see in the next section, DFDs are often partitioned into pieces; thus, we may find ourselves looking at one piece of the system that appears to have read-only (or write-only) stores. Some other piece of the system, documented on a separate DFD, may have the compensating write-only (or read-only) activity. Very tedious consistency checking is required to verify that some part of the system reads the store, and 14 (cont.) that some part of the system writes it; this is an area where the automated systems analysis packages discussed in Appendix A are extremely valuable.
16. ↑ It's very appealing to project managers, also. A project manager on one large project was heard saying to her project team, "I want all of you to bubble on down to the next level of detail by the end of this week!"
17. ↑ In some cases, it may even be appropriate to include control stores or event stores. These are analogous to the concept of semaphores first introduced by Dijkstra in (Dijkstra, 1968). For additional details, see (Ward and Mellor, 1985).