CHAPTER 12: Entity-Relationship Diagrams

“Obviously, a man’s judgment cannot be better than the information on which he has based it. Give him the truth and he may still go wrong when he has the chance to be right, but give him no news or present him only with distorted and incomplete data, with ignorant, sloppy or biased reporting, with propaganda and deliberate falsehoods, and you destroy his whole reasoning processes, and make him something less than a man.”

— Arthur Hays Sulzberger
Address, New York State Publisher’s Association, 1948

IN THIS CHAPTER, YOU WILL LEARN:

1. Why data models are useful in systems analysis;
2. The components of an entity-relationship diagram;
3. How to draw an entity-relationship diagram; and
4. How to refine an initial entity-relationship diagram.

In this chapter, we explore a graphical notation for modeling data. The entity relationship diagram (also known as an ERD, or E-R diagram) is a network model that describes the stored data layout of a system at a high level of abstraction. It is quite different from the dataflow diagram, which models the functions performed by a system; and it is different from the state-transition diagram, which models the time-dependent behavior of a system.

Why should we be interested in a data model of a system? Primarily because the data structures and relationships may be so complex that we want to highlight them and examine them independently of the processing that will take place. Indeed, this is particularly true when we show our system model to higher-level executive users in an organization (e.g., vice-presidents or department managers who may not be interested in the day-to-day operational details of the system). Such users are often more concerned with the data: What data do we need to run our business? How are the data related to other data? Who owns the data? Who is allowed to access the data?

Some of these questions, access to data, and ownership of data, for example, may be the responsibility of a dedicated group within the organization. The data administration group (or DA group) is often responsible for managing and controlling the essential information of the business; whenever you begin building a new information system, you will need to talk with these people so that you can coordinate your system’s information with their global, corporate-wide information model. The entity-relationship diagram is a useful modeling tool for carrying out this conversation.

There is often another group within the organization with a similar name: the database administration group (sometimes known as the DBA group). The group is usually located within the IT department (while the data administration group is not necessarily so located), and its job is to ensure that computerized databases are organized, managed, and controlled effectively. Thus, they are often the implementation team responsible for taking an essential model (one that is independent of a specific technology) and translating it into an effective, efficient physical database design for DB2, Oracle, Sybase, Ingres, IMS, ADABAS, or some other database management system. The entity-relationship diagram is an effective modeling tool for communicating with the DBA group. Based on the information presented by the ERD, the database administration group can begin to see what kind of keys or indexes or pointers they will need to access database records efficiently.

For the systems analyst, the ERD has a major benefit, too: it highlights relationships between data stores on the DFD that would otherwise be seen only in the process specification. For example, a typical ERD is shown in Figure 12.1. Each of the rectangular boxes corresponds to a data store on a DFD (a correspondence that we will explore further in Chapter 14), and you can see that there are relationships (connections) that one would normally not see on a DFD. This is because the DFD focuses the attention of the reader on the functions that the system is performing, not the data that it needs.

Indeed, consider an extreme case: what if there are no functions to be performed? What if the purpose of the system you are building is not to do anything, but merely to be the repository of a large amount of interesting information? Such a system might be called an ad hoc inquiry system, or a decision support system. In such a system, we might concentrate entirely on the data model, and not even bother building the function-oriented DFD model. Of course, this is indeed a rare situation: most systems do have functions to carry out; often we find that building the data model first makes it easier to discover what the required functions are.

Of course, the notation of the ERD in Figure 12.1 is quite mysterious at this point. In the following sections, we will examine the structure and components of an ERD; we will then discuss guidelines for drawing a well-structured ERD. The notation presented in this
12.1 THE COMPONENTS OF AN ERD

There are four major components of an entity-relationship diagram:

1. Object types
2. Relationships
3. Associative object type indicators
4. Supertype/subtype indicators

12.1.1 Object Types

An object type is represented by a rectangular box on an entity-relationship diagram; an example is shown in Figure 12.2. It represents a collection or set of objects (things) in the real world whose individual members (or instances) have the following characteristics:

- Each can be identified uniquely in some fashion. There is some way of differentiating between individual instances of the object type. For example, if we have an object type known as CUSTOMER, we must be able to distinguish one customer from another (perhaps by an account number, by last name, or by Social Security number). If all customers are the same (if we are operating a business where customers are just nameless, faceless blobs who come into our store to buy things), then CUSTOMER would not be a meaningful object type.
- Each plays a necessary role in the system we are building. That is, for the object type to be legitimate, we must be able to say that the system could not operate without access to its members. For example, if we are building an order entry system for our store, it might occur to us that in addition to customers, the store has a staff of janitors, each of whom is individually identified by name. While janitors presumably play a useful role in the store, the order entry system can happily function without them; therefore, they do not deserve a role as an object type in our system model. Obviously, this is something that you must verify with the users as you build your model.
- Each can be described by one or more data elements. Thus, a CUSTOMER can be described by such data elements as name, address, credit limit, and phone number. Many database textbooks describe this as attributing data elements to an object type. Note that the attributes must apply to each instance of the object type; for example, each customer must have a name, address, credit limit, phone number, and so on.

In many of the systems you develop, object types will be the system’s representation of a material thing in the real world. Thus, typical objects are customers, inventory items, employees, manufactured parts, and the like. The object is the material thing in the real world, and the object type is the system representation. However, an object may also be something nonmaterial: schedules, plans, standards, strategies, and maps are but a few examples.

Since people are often object types in a system, keep something else in mind, too: a person (or, for that matter, any other material thing) could be several different object types in different data models or even within the same data model. John Smith, for example, may be an EMPLOYEE in one data model and a CUSTOMER in a different data model; he could also be an EMPLOYEE and a CUSTOMER within the same data model.

Note that in all the examples of an object, we have used the singular form of a noun (e.g., employee, customer). This is not required, but it is a useful convention; as we will see in Chapter 14, there is a correspondence between objects in the ERD and stores in the DFD; thus, if there is a CUSTOMER object in the ERD, there should be a CUSTOMERS store on the DFD.

12.1.2 Relationships
Objects are connected to one another by relationships. A relationship represents a set of connections between objects and is represented by a diamond. Figure 12.3 shows a simple relationship that could exist between two or more objects.

![Figure 12.3: A relationship](image)

It is important to recognize that the relationship represents a set of connections. Each instance of the relationship represents an association between zero or more occurrences of one object and zero or more occurrences of the other object. Thus, in Figure 12.3, the relationship labeled **PURCHASES** might contain the following individual instances:

- Instance 1: customer 1 purchases item 1
- Instance 2: customer 2 purchases items 2 and 3
- Instance 3: customer 3 purchases item 4
- Instance 4: customer 4 purchases items 5, 6, and 7
- Instance 5: customer 5 purchases no items
- Instance 6: customers 6 and 7 purchase item 8
- Instance 7: customers 8, 9, and 10 purchase items 9, 10, and 11
- Etc.

As you can see, then, a relationship can connect two or more instances of the same object.

Note that the relationship represents something that must be remembered by the system — something that could not be calculated or derived mechanically. Thus, our data model in Figure 12.3 indicates that there is some important user-related reason for remembering the fact that customer 1 purchases item 1, and so on. And the relationship also indicates that there is nothing a priori that would have allowed us to determine that customer 1 purchased item 1 and no other items. The relationship represents system memory [1]. (An object represents system memory, too, of course.)

Note also that there can be more than one relationship between two objects. Figure 12.4, for example, shows two different relationships between a **PATIENT** and a **DOCTOR**. At first glance, you might think this is belaboring the obvious: every time the doctor treats a patient, he invoices the patient. But Figure 12.4 suggests that the situation might be different: it may turn out, for example, that there are several different instances of a “treatment” between a doctor and the same patient (i.e., an initial treatment, follow-up treatments, etc.). And Figure 12.4 implies that the invoicing relationship is entirely separate from the treatment relationship: perhaps some patients are invoiced only for their first treatment, while others are invoiced for every treatment, and still others are not invoiced at all.

![Figure 12.4: Multiple relationships between objects](image)

A more common situation is to see multiple relationships between multiple objects. Figure 12.5 shows the relationship that typically exists between a buyer, a seller, a real estate agent, the buyer’s attorney and the seller’s attorney, for the sale and purchase of a house.

![Figure 12.5](image)
With a complex diagram like the one in Figure 12.5 (which is typical of, if not simpler than, the ERDs you are likely to find in a real project), the relationship and its connected object types should be read as a unit. The relationship can be described from the perspective of any of the participating object types, and all such perspectives are valid. Indeed, it is the set of all such viewpoints that completely describes the relationship. For example, in Figure 12.5, we can read the negotiates price relationship in any of the following three ways:

1. Real estate agent negotiates price between buyer and seller.
2. Buyer negotiates price with seller, through real estate agent.
3. Seller negotiates price with buyer, through real estate agent.

Note that, in some cases, we can have relationships between different instances of the same object type. For example, imagine a system being developed for a university, in which COURSE, STUDENT, and TEACHER are object types. Most of the relationships that we will concentrate on are between instances of different object types (e.g., the relationships “enrols in,” “teaches,” etc.). However, we might need to model the relationship “is a prerequisite for” between one instance of COURSE and another instance of COURSE.

### 12.1.3 Alternative Notation for Relationships

As we have seen in Section 12.1.2, relationships in the E-R diagram are multidirectional; they can be read from any direction. Also, we have seen that the E-R diagrams do not show cardinality; that is, they do not show the number of objects participating in a relationship. This is quite conscious and deliberate: we prefer to bury such details in the data dictionary. This is discussed further in Section 12.3.

An alternative notation used by some systems analysts shows both cardinality and ordinality. For example, Figure 12.6(a) shows a relationship between CUSTOMER and ITEM in which the additional notation indicates that

1. The CUSTOMER is the anchor point, the primary object from whose viewpoint the relationship should be read.
2. The relationship consists of one customer connected to N items. That is, an individual customer can purchase 0, 1, 2, ... or N items. However, the relationship indicates that only one customer can be involved in each instance of a relationship. This would preclude, for example, the case where multiple customers are involved in the purchase of a single item.

Another common notation is shown in Figure 12.6(b); the double-headed arrow is used to show the one-to-many relationship, while a single-headed arrow is used to show a one-to-one relationship between objects.

Such annotated E-R diagrams are discussed in detail in [Chen, 1976], [Martin, 1982], and [Date, 1986]. However, I prefer not to include such detail, for it can easily be included in the data dictionary (as discussed in Section 12.4), and it tends to distract from the major purpose of the E-R diagram, which is to give an overview of the components of and interfaces between the elements of data in a system. Though there is nothing intrinsically wrong with procedural annotations on the diagram, my experience has been that systems analysts often carry a good idea too far and clutter up the diagram with far more information than is appropriate.

### 12.1.4 Associative Object Type Indicators

A special notation on the E-R diagram is the associative object type indicator; this represents something that functions both as an object and a relationship. Another way of thinking about the associative object type is that it represents a relationship about which we wish to maintain some information.
Consider, for example, the simple case of a customer purchasing an item (or items), which we illustrated in Figure 12.6. Regardless of whether we include the procedural annotation or not, the main point is that the relationship of PURCHASE does nothing more than associate a CUSTOMER and one or more ITEMS. But suppose that there is some data that we wish to remember about each instance of a purchase (e.g., the time of day when it took place). Where could we store this information? “Time of day” is certainly not an attribute of CUSTOMER, nor is it an attribute of ITEM. Instead, we attribute “time of day” to the purchase itself and show it in a diagram as illustrated by Figure 12.7.

![Figure 12.7: An associative object type indicator](image)

Notice that PURCHASE is now written inside a rectangular box and that it is connected, via a directed line, to an unnamed relationship diamond. This is meant to indicate that PURCHASE functions as:

- **An object type**, something about which we wish to store information. In this case, we want to remember the time at which the purchase took place and the discount offered to the customer. (Again, this assumes that such information could not be derived, after the fact, by the system.)
- **A relationship** connecting the two object types of CUSTOMER and ITEM. The significant thing here is that CUSTOMER and ITEM stand on their own. They would exist whether or not there was a purchase. [4] A PURCHASE, on the other hand, obviously depends for its very existence on the CUSTOMER and the ITEM. It comes into existence only as the result of a relationship between the other objects to which it is connected.

The relationship in Figure 12.7 is deliberately unnamed. This is because the associative object type indicator (PURCHASE) is also the name of the relationship.

12.1.5 Subtype/Supertype Indicators

The subtype/supertype object types consist of an object type and one or more subcategories, connected by a relationship. Figure 12.8 shows a typical subtype/supertype: the general category is EMPLOYEE and the subcategories are SALARIED EMPLOYEE and HOURLY EMPLOYEE. Note the subtypes are connected to the supertype via an unnamed relationship; note also that the supertype is connected to the relationship with a line containing a crossbar.

![Figure 12.8: A supertype/subtype indicator](image)

In this notation, the supertype is described by data elements that apply to all the subtypes [5]. For example, in Figure 12.8, we could imagine that all employees are described by such facts as:

- Name
- Years of service
- Home address
- Name of supervisor

However, each subtype is described by different data elements; otherwise, there would be no point distinguishing between them. For example, we could imagine that a SALARIED EMPLOYEE is described by such things as:

- Monthly salary
- Annual bonus percentage
- Amount allowed for company car

And the HOURLY EMPLOYEE might be described by such things as:

- Hourly pay
- Overtime rate
- Starting time
12.2 GUIDELINES FOR CONSTRUCTING ENTITY-RELATIONSHIP DIAGRAMS

The notation shown in the previous section is sufficient to build arbitrarily complex entity-relationship diagrams. However, you may be thinking to yourself at this point, “How do I discover what the objects and relationships are in the first place?” Your initial model of object types and relationships will usually be derived from (1) your understanding of the user’s application, (2) interviews with the user, and (3) any other research and data gathering that you can use. (Interviewing and data gathering techniques are discussed in Appendix E.)

You should not expect that the first E-R diagram you draw will be a final one that you will review with the user community or deliver to the system designers. Like dataflow diagrams and all the other systems modeling tools, E-R diagrams should be revised and improved several times; the first version will typically be nothing more than a crude beginning, and subsequent versions will be produced using a series of refinement guidelines presented in this section. Some of the refinement guidelines will lead to the creation of additional object types, while other refinement guidelines will lead to the removal of object types and/or relationships.

12.2.1 Adding Additional Object Types

As indicated above, your first-cut ERD will typically be created from initial interviews with the user and from your knowledge of the subject matter of the user’s business. This will, of course, give you a strong clue as to the identity of the major objects and relationships [6].

After the first-cut ERD has been developed, the next step that you should carry out is that of attributing the data elements in the system to the various object types. This assumes, of course, that you know what the data elements are! This may happen in any one of three ways:

1. If the process model (DFD) has already been developed or is being developed in parallel with the data model, then a data dictionary will already exist. It may not be complete at this point, but it will be enough to begin the process of attribution.
2. If the process model has not been developed (or, in the extreme case, if you don’t intend to develop one at all), then you may have to begin by interviewing all the appropriate users to build up an exhaustive list of (and definitions of) data elements. As you do this, you can attribute the data elements to the objects in the E-R diagram. (However, note that this is a time-consuming, bottom-up process that may cause some delays and frustration.)
3. If you are working with an active data administration group, there is a good chance that a data dictionary will already exist. This may be given to you at a fairly early stage in your project, at which point you can begin the process of attribution.

The attribution process may provide one of three reasons for creating new object types:

1. You may discover data elements that can be attributed to some instances of an object type, but not to others.
2. You may discover some data elements that are applicable to all instances of two different objects.
3. You may discover that some data elements describe relationships between other object types.

If, during the process of attributing data elements to object types, you find that some data elements cannot apply to all instances of an object type, then you will need to create a set of subtypes beneath the object type you have been working with and assign the unique data elements to appropriate subtypes.

Suppose, for example, you are developing a personnel system, and you have identified (with extraordinary brilliance and creativity!) an object type called EMPLOYEE. While reviewing the available data elements, you find that many of them (age, height, date-employed, etc.) are applicable to all instances of an employee. However, you then discover a data element called number-of-pregnancies; it is obviously a relevant data element for female employees but not for male employees. This would prompt you to create MALE-EMPLOYEE and FEMALE-EMPLOYEE as subtypes of the general category of employee.

Obviously, I am not suggesting that all personnel systems should keep track of the number of pregnancies that each employee has had; the example was chosen merely because there is general agreement that male employees cannot be pregnant. Compare this, however, to the data element spouse’s-name: there are several instances of EMPLOYEE for whom spouse’s-name may not apply (because they are not currently married), but that is a very different situation from that of a data element that cannot apply [7].

In most cases, this process of creating new subtypes and assigning data elements appropriately is very straightforward. However, keep one exceptional situation in mind: it may turn out that all the relevant data elements are attributed to one of the subtypes, and that none of the data elements can be attributed to the supertype object; that is, it may turn out that the data elements are mutually exclusive, belonging to one subtype or another subtype, but not both. Suppose, for example, that the only data elements that we can attribute to employees are number-of-pregnancies and years-of-experience-playing-on-New-York-Knicks-basketball-team. We might well decide (after wondering what kind of lunatic user was responsible for such a system!) that the general supertype of EMPLOYEE did not apply.

The opposite situation can occur, too: data elements may describe instances of two (or more) different object types in the same way. If this occurs, then you should create a new supertype and assign the common data elements to the supertype. For example, we may have identified CASH-CUSTOMER and CREDIT-CARD-CUSTOMER as two distinct object types when first creating an ERD for an order entry system (perhaps because the user told us that these were two distinct categories). However, it might quickly become apparent that the data elements customer-name and customer-address describe both types of customer in the same way; this would argue for the creation of a supertype, as shown in Figure 12.9.
Similarly, if a data element describes the interaction of two or more object types, then we should replace the “naked” relationship between the two objects with an associative object type. For example, look at the first-cut ERD shown in Figure 12.10(a) in which there is a PURCHASES relationship between CUSTOMER and ITEM. During the attribution of data elements, we might find that there is a data element called date-of-purchase that (1) seems to belong to the PURCHASE relationship, and (2) obviously describes, or provides data about, the interaction of a CUSTOMER and an ITEM. This suggests that we should replace the relationship PURCHASES with an associative object type, as shown in Figure 12.10(b).

Sometimes the initial E-R diagram will contain an object type that, on closer analysis, clearly deserves to be an associative object type. For example, Figure 12.11(a) shows an E-R diagram with three related objects: CUSTOMER, ORDER, and PRODUCT. During the process of attributing data elements to the various objects, we find that date-of-delivery most sensibly belongs to the object ORDER; after all, customers don’t get “delivered,” and products get delivered only as the result of an order. Indeed, thinking about this makes it evident that the ORDER itself is a relationship between CUSTOMER and PRODUCT, as well as an object about which we wish to remember some facts. This leads to Figure 12.11(b).

Finally, we have the case of repeating groups. Consider, for example, the object type EMPLOYEE, with such obvious data elements as name and home address. Now suppose that we find additional data elements child’s-name, child’s-age, and child’s-sex. It could obviously be argued that child’s-name, child’s-age, and child’s-sex are ways of describing a new object type called CHILD, which has
inadvertently been previously embedded within EMPLOYEE. It could also be argued that there are (potentially) multiple instances of child-related information associated with each instance of an employee, and that each instance of child-related information is uniquely identified by the child’s-name. In this case, the object type that we initially thought of in the form shown in Figure 12.12(a) should be transformed into two object types, connected by a new relationship, as shown in Figure 12.12(b).

This process of removing embedded objects is part of a more comprehensive refinement activity often described as normalization. The objective of normalization is to produce object types in which each instance (or member) consists of a primary key value that identifies some entity, together with a set of mutually independent attribute values that describe that entity in some way. The process of normalization is described in detail in Chapter 14 of [Date, 1986] and in Chapter 19 of [DeMarco, 1978].

12.2.2 Removing Object Types

The previous section dealt with ERD refinements that created additional objects and/or relationships. However, there are also a number of situations where refinement of the ERD will lead to the removal of redundant or erroneous object types and relationships. We will examine four common situations:

1. Object types that consist of only an identifier
2. Object types for which there is only a single instance
3. Dangling associative object types
4. Derived relationships

If we have an E-R diagram in which one of the object types has only an identifier assigned to it as a data element, there may be an opportunity to eliminate the object type and assign the identifier as a data element in a related object type. For example, let us imagine that we have constructed an initial E-R diagram as shown in Figure 12.13(a) for a personnel system:
During the process of attributing data elements to the various objects, though, we may find that the *only* information that the personnel system keeps about a spouse is his or her name (i.e., the identifier that distinguishes one spouse from any other spouse in the system). In this case, an obvious refinement is to eliminate *SPOUSE* as an object type and instead include spouse-name as a data element within the object *EMPLOYEE*.

Note that this refinement only makes sense when there is a one-to-one correspondence between instances of the about-to-be-deleted object and instances of the related object. The example immediately above, for instance, makes sense because modern society presumes that a person will have, at most, one spouse. This leads to the reduced E-R diagram shown in Figure 12.13(b).

An even greater reduction can be made if we find that our initial E-R diagram contains an object for which our only fact is the identifier and that is a single-instance object. Consider the initial E-R diagram shown in Figure 12.14(a).

At first glance, this seems like a reasonable way of showing the relationship between patients and drugs (medicinal, of course!) in a hospital. But suppose the only information that we keep about the drug is its name (identifier); and suppose that the hospital only administers one type of drug (e.g., aspirin). In this case, the drug is a constant and does not even have to be shown on the diagram. (Note that this also means that our system would not have a data store called drugs). Our reduced diagram would look like Figure 12.14(b).
Because of the situation above, it is possible to create a dangling associative object type. Consider the following variation on the previous hospital example, as shown in Figure 12.15(a). If, as suggested above, it turns out that DRUG is a single-instance, identifier-only object, then it would be deleted. This would result in the reduced diagram shown in Figure 12.15(b); note that TREATMENT is still an associative object type, even though it is connected to only one other object type. This is known as a dangling associative object type and is quite legal (though somewhat unusual) in an ERD.

![Figure 12.15(a): An initial E-R diagram](image)

![Figure 12.15(b): The reduced E-R diagram](image)

Finally, relationships that can be derived, or calculated, should be removed from the initial E-R diagram. As mentioned earlier in the chapter, the ERD should show the requirements for stored data. Thus, in Figure 12.16(a), if the renews relationship between DRIVER and LICENSE can be derived (based on the driver’s birthday, or on the first letter of his last name, or on a variety of other schemes often used by Motor Vehicle Bureaus throughout the United States), then it should be eliminated. This leads to Figure 12.16(b), in which the object types are unconnected. This is quite legal in an ERD; it is not necessary for all object types to be connected to one another.

![Figure 12.16(a): An initial ERD](image)

![Figure 12.16(b): The reduced ERD](image)

### 12.3 EXTENSIONS TO THE DATA DICTIONARY FOR E-R DIAGRAMS

Finally, we observe that the data dictionary discussed in Chapter 10 needs to be extended to provide for the ERD notation discussed in this chapter. In general, objects on the ERD correspond to stores on the DFD; more about this will be discussed in Chapter 14. This means that in the data dictionary definition below, CUSTOMER is both the definition of the object type and an instance of the store CUSTOMERS.
CUSTOMERS = \{CUSTOMER\}

CUSTOMER = @customer-name + address + phone-number

Note also that the definition of a CUSTOMER includes a specification of the key field, which is the data element (or attribute) that differentiates one instance of a customer from any other. The "at" sign (@) is used to indicate the key field(s). [8]

However, we also need to include in the data dictionary a definition of all the relationships shown in the ERD. The relationship definition should include a description of the meaning of the relationship, within the context of the application; and it should indicate the objects that form the association. Appropriate upper and lower limits should be specified to indicate whether the association is a one-to-one association, a one-to-many association, or a many-to-many association. For example, the relationship purchases shown in Figure 12.10(a) might be defined in the data dictionary in the following way:

\[
purchases = \{\text{the association of a customer and one or more items}\}
\]

\[
\text{@customer-id} + 1\{\text{item-id + quantity-purchased}\}
\]

12.4 SUMMARY

For any system with multiple stores (objects) and complex data relationships, the ERD can be a valuable tool. As we have seen in this chapter, it focuses entirely on the data relationships, without providing any information about the functions that create or use the data.

While we have used the ERD in this book as the graphic modeling tool for showing data relationships, you should be aware that a variety of other modeling tools are used for the same purpose; [Martin, 1982] and [Date, 1986] show many examples of alternative modeling tools.

At this point, many students ask whether the DFD model should be developed first and then the ERD, or whether it is better to develop the ERD first and then the DFD. Indeed, some students even ask whether it is really necessary to develop both models, when either the DFD or the ERD provides so much interesting information. The answer to the first question is simple: it doesn't matter which model gets developed first. Depending on your own preferences or the preferences of the user, or the nature of the system itself (i.e., whether it is function-rich or data-rich), one of the models will often cry out to be developed first. In other cases, though, you may find that both models are developed concurrently; this is particularly common when the project team contains a distinct database design group, or when the EDP organization has a data administration group that develops corporate data models.

The second question is more important: Is it really important to develop two different models of a system (and, as we will see in Chapter 13, a third model of the time-dependent behavior of the system)? The answer is that it depends on the kind of system you are developing. Many of the classic business data processing systems developed in the 1960s and 1970s appeared (at least superficially) to consist of many complex functions, but relatively trivial data structures; hence the DFD model was emphasized, and the ERD model was often ignored. Conversely, many of the decision-support systems and ad hoc database inquiry systems that began being built in the 1980s appeared (at least superficially) to consist of complex data relationships, but almost no functional activities; hence the ERD model was emphasized, and the DFD model was downplayed. And the timing characteristics of real-time systems built in the 1960s and 1970s appeared (at least superficially) to dominate any consideration of functions and data relationships; in such systems, the state-transition model (discussed in Chapter 13) was often emphasized to the exclusion of DFDs and ERDs.

By the mid-1990s, though, many IT organizations found that they were building systems that were far more complex than the special-purpose systems of a decade or two earlier; indeed, many of them were between 100 times and 1000 times larger and more complex. Several of these large, complex systems had incredibly complex functions and complex data relationships and complex time-dependent behavior; consider, for example, the Star Wars missile-defense system, which was estimated to require 100 million computer instructions, and which had a real-time behavior of mind-boggling complexity [9]. For any such complex system, it is obvious that all three of the modeling tools discussed in this book will be critically important. On the other hand, if you do become involved in a simple, one-dimensional system, you may find that you can concentrate on the modeling tool that illuminates and highlights the one important aspect of the system.

In Chapter 14, we will see how the ERD, the DFD, the state-transition diagram, the process specification, and the data dictionary can all be checked against each other in order to produce a complete system model that is internally consistent.

REFERENCES


QUESTIONS AND EXERCISES