Creating a Logical Data Model

Terms you'll need to understand:

- ✓ Entity Relationship Model
- ✓ Entity
- ✓ Kernel entity
- ✓ Associative entity
- ✓ Characteristic entity
- ✓ Attribute
- ✓ Identifier
- ✓ Dependency
- ✓ Primary key
- ✓ Foreign key
- ✓ Normalization
- ✓ Denormalization
- ✓ Normal forms
- ✓ Relationship
- ✓ Cardinality

Techniques you'll need to master:

- ✓ Entity Relationship Modeling
- ✓ Database normalization/denormalization
- ✓ Key and relationship definition

Introduction

Designing a logical data model is all about preparing for the physical data design. Whether physical elements end up being a simple database, a complex data warehouse, or some other data store, the ideas behind the logical design remain the same. In this stage of development, try not to gear any design to a specific physical structure. It is important to keep in mind that we are still just planning things out, and the decisions regarding the physical elements have yet to be made.

Even though the physical elements have yet to be determined, there is a definite correlation between what you see in the logical model and what will end up being present when the model takes on a physical form. There is almost a one-to-one mapping between the objects that we discuss in the logical stage of development and similar objects in the physical databases, tables, files, and other physical articles.

Though there are many data storage possibilities, the largest percentage of these will be databases—the storage mechanism of choice for most systems. Relational databases are databases in which data is organized into related objects. Each of the objects contained in a database is related to the others in some way.

Relational databases, based on the paper written by Dr. E. F. Codd in 1970, store sets of data in relations, called *tables*. The tables are often related to one another through dependencies, but this is not required. In all the time that has passed since then, the modeling of data structures has remained based on this concept.

Data Modeling and Logical Data Design

Most individuals in the technology arena understand the physical components of a database system. It is easy to recognize data in a columnar format from spreadsheets, tables, data files, and other common data storage techniques. In a structured data storage system, however, much planning goes into the makeup of the storage system before any physical forms are taken. Long before a combination of files containing records and fields along with tables containing rows and columns form a database, the data content is analyzed and a concept of the data, or *logical structure* is formed. This is an

involved process that takes much practice before a database professional can become competent. *Logical modeling* is an integral part of database implementation. For this reason, logical data modeling encompasses many of the objectives of the exam.

It is difficult to discuss the data modeling exam objectives without looking at some of the physical elements in the database itself. This chapter mentions and defines some of the physical components of a database system. However, in this chapter our focus will be the logical design of the database. In most business problems the physical database itself would not be considered until a logical model had been drawn. For this reason, the physical elements mentioned in this chapter are discussed in full in Chapter 3, "Implementing the Physical Database," so we can concentrate here on drawing logical data models.

Logical data modeling in practice involves identifying important elements of data and recognizing how these elements interact. It is also important to be knowledgeable about the business process being analyzed. This knowledge will aid in determining data flows and processing required to pull meaningful information out of reams of raw data.

For those heading down the path to becoming a database administrator, data modeling plays an important role in the MCDBA 70-229 Exam, but for those attempting the MCSD certification, similar information may be tested in the MCSD Exam 70-300 .NET Solutions Architecture Exam.

In designing a database, to meet a given business need, the logical data model most used is the *Entity Relationship Model*, or *ER Model*. In determining the ER model, you define the elements of data used in the business process and how they relate to each other. In looking at any logical implementation, you must understand the nature and use of the data. Knowing how the data is used helps you understand the relationships between the separate data elements.

Source documents, reports, and other samplings from an existing business process, together with information gathered through interviews, will assist in collecting data examples. With this information you can begin designing a data model in accordance with the current business situation. It is important that in gathering information, the data samples collected and drawn up be as complete as possible. Accompanying these data samples, full descriptions of all procedures that interact with the data would also be used as an aid to the development of a model.

Data Modeling

To develop a data model, various application architectures can be considered. During logical modeling, variations in implementations do not need to be known. In meeting a specific business need, the data needs to be modeled in a pristine fashion. If you have a tendency toward one implementation or another, you could hamper this design goal. It is best to design the model as a generic data model before taking that model into future phases. In subsequent stages of development, the database will take on its physical attributes and will be adjusted as applications take shape; yet these stages should not be stepped into prematurely. Focus first on the raw information as a base for the logical model.

Relational database design modeling was first developed by the database engineer Charles Bachman, in 1960, and then in 1976, database design modeling became the ER Model by Peter Chen. An ER Model allows a database to be defined in a simple and organized manner. Other modeling techniques have come and gone over the years, but the ER Model is the preferred technique used by most experienced database developers.

Modeling with the ER Data Model

As mentioned previously, the ER data model is one of the most popular data models in use. An ER data model consists of three main components that are present in any model format: *entities*, *attributes*, and *relationships*.

- An *entity* is a discernible thing about which data is kept. In data modeling, entities can be closely compared to the physical element of a table in a database. An entity is a collection of related data elements or attributes and usually represents a major facet of the business problem. Supplier, product, employee, and order are all good examples of entities.
- *Attributes* are the characteristics given to an entity, such as `ProductNumber` and `FirstName`. Attributes of an entity represent a set of properties, each property being a data element within an entity. They will become the columns or fields in the physical database.
- *Relationships* show a logical link between two or more entities. Where two entities have a direct affiliation, a relationship is used to define the connection between the entities. A supplier entity may be related to orders; a product entity may be related to purchases. Relationships will normally establish constraints within the physical database.

Entities represent the primary elements of the ER Model. Each entity will represent a person, place, thing, or concept involved in a business process. It is usually easiest to determine the entities of a process by isolating the important players. When a customer places an order containing one or more products, the process utilizes at least three major entities: *customer*, *order*, and *product*.

An ER Model is drawn out using an *Entity Relationship Diagram (ERD)*. A rough diagram can be constructed using pencil and paper and a few simple shapes. For a more polished look a drawing tool is recommended, such as Microsoft Paint, or better yet a tool specifically used for the modeling process, such as Microsoft Visio. The actual tool used for drawing the diagram can vary, and you will find that different developers have their own preferences and reasons for preferring one tool over another. Most of the diagrams throughout this book have been developed using the two previously mentioned programs, along with SQL Server's own built-in tools. To illustrate the concepts of the ER modeling process, we will be using a fictional company, *Northwind Traders*. This company will be familiar to many Microsoft Access users and it is also supplied with SQL Server as a sample database.

Entity Selection

To begin the modeling process, you need to first isolate the entities required for the business process. A standard documentation technique is to draw entities as rectangular boxes with enough space to hold many attributes. An entity is the basic division of a database. In the logical design, entities are representative of the tables that will be present when the database development process moves into the physical design phase. Each entity exists as a separate individual data collection, unique from all the other entities.

Entities are generally the people, places, and things that make up a process. They can be qualified as one of three basic types: *kernel entities*, *associative entities*, and *characteristic entities*. These entity types are described further in the following list:

- A *kernel entity* exists on its own; it doesn't define or provide descriptive information for other entities. An example of a kernel entity would be a product listing in an inventory model. The information contained in each kernel entity of a table represents the heart of the database model.
- *Associative entities* are needed to allow multiple kernel entities to be tied together. In the inventory system, a sales entity would be needed to tie a customer kernel entity to the products they have purchased. This same sales entity could be tied to another kernel entity, such as salespeople.

- A *characteristic entity* provides additional information for a respective kernel or associative entity. Information contained in characteristic entities can be updated independently of the related entity. A product entity could have a characteristic parts entity. A given product could be made up of a number of parts. A part that becomes unavailable could affect the product's availability. Changes over time to parts information could be made more easily if a parts entity existed, instead of your having to make changes against the products kernel.

Some entities stand out within a process and are easily recognized, particularly those that represent people or organizations. Entities such as customer, supplier, employee, and shipper are all relatively easy to identify, whereas other entities are more difficult to identify. Careful thought about a business process will help flush them out. Let's look at the process used when a customer orders merchandise and use it as an example.

A customer will order products from one of our salespeople. The order is recorded on an invoice on which each line item represents the quantity of a single product ordered. Any products not currently in stock will be backordered. Ordering the needed products from the supplier will fill backorders. When an order is ready, the company can ship it to the customer using one of the available shipping methods. This simple process indicates that a few other entities are needed. Order, product, and order detail will be needed to fulfill the order process and track the data accordingly.

This process will allow for the initial sketch of entities to be drawn, as shown in Figure 2.1.

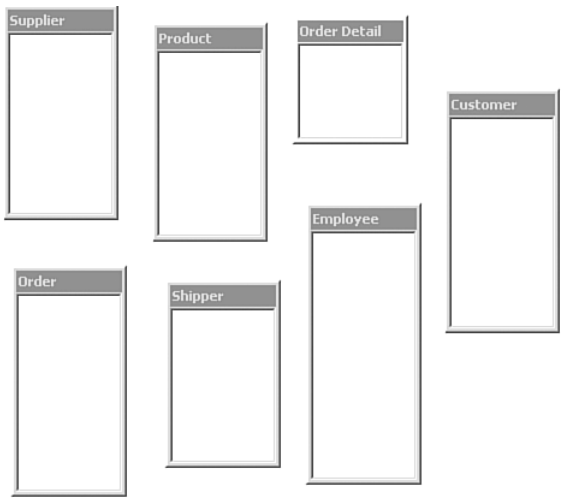

Figure 2.1 A rough draft of Northwind's entities.

With further knowledge of the business process, other entities may come to light. In the case of the Northwind process, products all fall into separate categories and the product categories are also tracked. This would require the addition of an additional entity, and the beginnings of the ER diagram would look similar to the example shown in Figure 2.2.

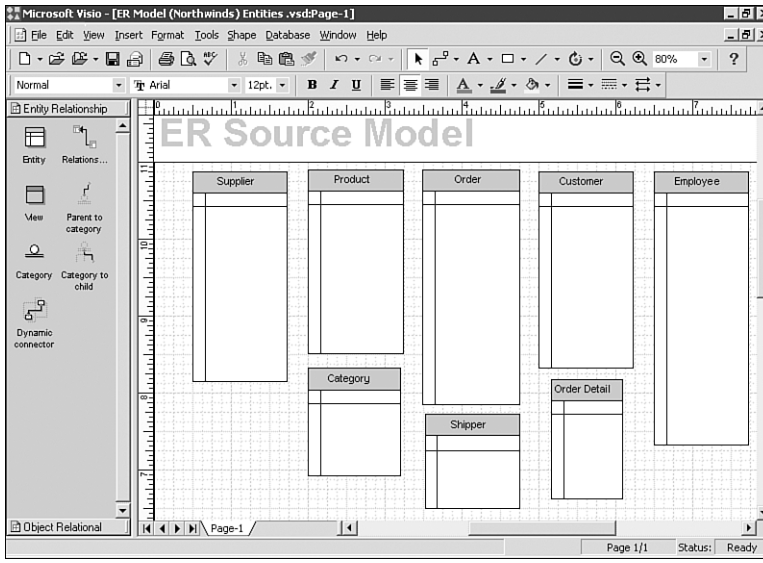

Figure 2.2 Northwind's entities in an ER Model.

Entity structuring accommodates the initial stages of database design. When you're designing an appropriate logical model, the data must be organized into these separate elements that will later make up the physical database tables. An entity is characterized by its attributes. Attributes are used to define the data elements of an entity. After the initial entities have been defined, the process of describing each entity through its characteristic properties begins.

Attribute Definition

Identifying attributes is the next step in ensuring a successful data modeling process. In defining attributes you are setting out to define entity composition. Each entity will have descriptive elements that pertain solely to that element. An attribute is a descriptive element or property of an entity. Fields will represent the attributes when the logical design progresses to the physical design stage.

Attributes are characteristics or properties defined within a single entity, and they correspond to real-world properties of a place, thing, or concept.

Attributes such as names and addresses are almost always present for people and organizations. Other attributes provide further information for the entity as required for the business process being defined.

Deciphering attributes from written descriptions and reports is more of a real-world scenario. The exam will provide the attributes; you will be expected to pick appropriate entities.

Try to find out the attributes that fit each of the entities. More attributes may have to be added later as the model becomes more complete. If you missed an attribute or added extra attributes at this time, they will likely be pointed out when the model is normalized. The normalization of a data model will be discussed later in this chapter, in the section "Data Normalization with Normal Forms." Attribute decisions will vary from person to person, depending on your business scenario perspective for which the data is being modeled.

The identification and creation of attributes is a developed skill; there is no true method for defining all attributes of an entity. Each business problem will require a variation of entity content, so the business process itself will lead to a lot of attribute choices.

A few guidelines to use in the identification, creation, and naming of attributes will help ease this process. The first is how you name your entities. A good name makes an attribute look professional and helps in its readability. Appropriate naming conventions are often developed as a corporate or development standard within an organization. Often mechanisms for shortening names or using common prefixing or suffixing is part of a programming team's standard. Here are some good guidelines that help in naming entities. Consistently following these guidelines will help to keep all of your designs up to the same standard:

- An entity or an attribute should be named in its singular form, thereby implying that it is only a single instance. An *instance* is a single occurrence of an entity.
- The use of underscores (_), spaces, and other special characters is not a good habit because special characters have particular meanings in some software packages, and the mixture of text and other characters is difficult to type. Try to distinguish a word from another by using mixed case, as in `LastName` instead of `Last_Name`.

- Entity (and all other object name) identification should be kept small while still providing a description of the object. Names should be kept as small as possible but should still provide a meaningful object title.
- Entity names should be unique.
- Reserved words, though permitted in the context of SQL names, should be minimized to ease development. Later this will also add to the performance of procedures.

Keep in mind that many of these guidelines refer to all object naming, and when developing the names for attributes, you should still be providing a descriptive name that is concise and unique within the entity. Attribute names should be consistent across entities. For example, if you name an attribute `LastName` within one entity, you should not name a similar attribute `Surname` in other entities.

Decomposing an Attribute

Many attributes can be handled as a single attribute. It is also common for some attributes to be broken down into other, smaller attributes. This process is commonly known as *decomposing* attributes. Decomposing an attribute takes an attribute from its original form and divides it into its components. A good example of this is the breaking down of the `Address` attribute. An `Address` attribute can easily be broken down into attributes that store data as shown here:

`Street`	Stores the street address of the user
`City`	Stores where the user lives
`Region`	Stores the state or province the user lives in
`Postal Code`	Stores the user's zip code or other postal code
`Country`	Stores the user's nation

The process of decomposing an attribute helps to develop a normalized structure, as defined later in this chapter. Decomposing is a function of usage as well. If, for example, a person's name is needed only in its full form, then a decision may be made to not break it up into the separate attributes of first name and last name. This is common for a `ContactPerson` attribute in an entity that relates to a corporation.

Decomposing an attribute provides many benefits, in contrast to generic compound attributes that are not decomposed. Data integrity is improved. *Data integrity* is a measurement of how well data is kept consistent and

flawless throughout the whole ER Model. When attributes are decomposed, different methods of ensuring data integrity can be applied to the broken-down segments rather than the attribute as a whole. It can be difficult to check the validity of an entire address, but when decomposed, the elements can be more easily checked.

Decomposing also aids in the sorting of data for use in specific business processes, such as mass mailing. You will, in most cases, also be improving data retrieval performance when decomposed attributes are used. A generic attribute, `Address`, contains the street, city, region, postal code, and country. To get just the region of a customer in Washington, you have to select the whole `Address` attribute and parse it to find `Washington`, thereby degrading performance because of the redundant data retrieved. If you have four separate attributes, you can select the `Region` column and get your results more quickly. This same rule applies to updating data. It's easier to update a single part of an `Address` than to parse and then update the whole `Address` attribute.

As shown by the example provided in Figure 2.3, the `Address` has been fully decomposed for all entities, but the name has been decomposed only for the `Employee` entity.

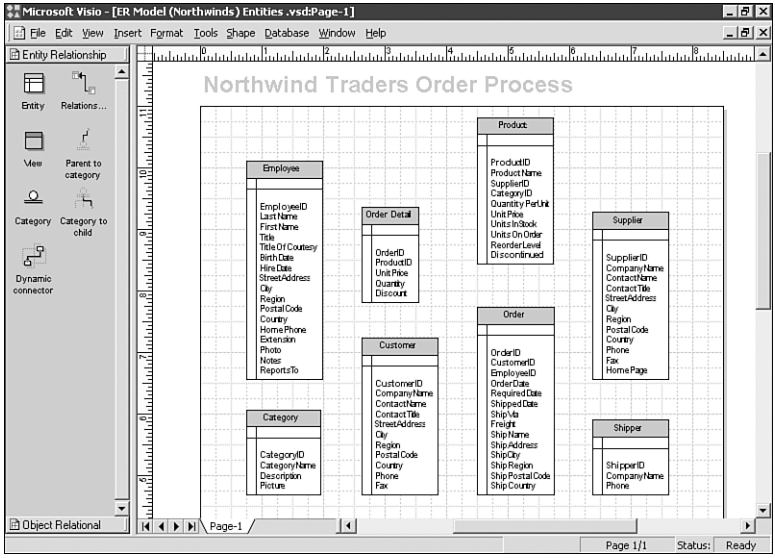

Figure 2.3 Northwind's entities with attributes.

Key Attributes

The use of an attribute can vary from system to system, but some attributes will be present in most systems to help sort data and perform relationship ties

between one entity and another. A *key attribute* is almost always present within an entity to act as an identifier, much as a person's name identifies that person as being a unique individual. Entities are usually interdependent: Each holds information that relates to other entities. These relationships can be defined by their correlated dependencies. Key attributes are also therefore used for the purpose of relating one entity to another.

Recognizing Key Attributes

After all attributes have been defined and keys have begun to be recognized, the modeling process will be completed with the application of relationships and the normalization of data. These two processes are closely related, as you will see later, but before they can begin, key attributes must be recognized. These are specialized attributes referred to as *identifiers*. An identifier is an attribute or a set of attributes that defines one unique element of an entity. The use of identifiers allows for the individual selection of records from an entity. As the design progresses to the physical stage, identifiers will become the *primary* and *foreign keys*, allowing entities to be tied together through association or relationships. For Example, a product's identifying attribute is usually a unique product ID.

Identification of Primary and Foreign Keys

A *primary key* is a specialized attribute that is generally defined for each entity. The primary key is almost always defined, though is not necessarily required for all entities in a data model. However, the provision of a primary key does allow for a considerable number of benefits and should be considered in every instance. When defining a primary key, you should keep various factors in mind. The primary key normally defines uniqueness in an entity in that every record of a table has its own unique primary key. Also, when defined, a primary key should not be permitted to be empty. If a primary key is empty, you have a situation in which data integrity is difficult (if not impossible) to maintain.

A primary key should be defined as a single attribute that doesn't allow for duplicates or empty content. The primary key should be as small as possible. It is possible to create a compound primary key that uses multiple attributes or a key that contains a large number of bytes, but in the physical design this will increase the overhead and response time associated with data retrieval. A compound primary key consisting of multiple attributes is used in instances in which a singular attribute by itself does not enforce uniqueness within an entity. As we move through the modeling process, we will discover the need to use such a compound structure in some instances.

In most cases, an attribute is identified for each entity that will act as a key. This attribute could be a customer number, a product number, or simply an attribute to serve no other purpose than to act as a key identifier. When an attribute is added solely for the sake of being an identifier, it is known as a *surrogate key*. Whether you use a natural or surrogate key, that key will represent an important part in establishing relationships. In most circles surrogate keys are preferred because there is never a need to have this surrogate key change. The process of altering a key value can have repercussions on the business process and can also effect many elements of the database system.

A longtime favorite answer on Microsoft database exams is to use surrogate keys in all entities. As mentioned previously, there are several benefits to doing this.

A *foreign key* is used to tie one entity to the primary key or unique data value of another entity. The relationship is created for the purpose of creating a dependency between the entities. A single attribute or combination of attributes can act as a foreign key depending on the makeup of the referenced primary key. A foreign key doesn't have to be unique. In fact, foreign keys are often in a many-to-one relationship with a primary key in another entity. Foreign key values* should be copies of the primary key values. No value in the foreign key, except a null value, should ever exist unless the same value exists in the primary key of the referenced entity.

A foreign key works in conjunction with a primary key or some other unique attribute to enforce referential integrity among entities. A relationship is created to enforce referential integrity between these two related entities. Foreign key connections may not be fully recognized for the model until you begin to look at the interrelationships of the entities. After a relationship is defined, the connection developed for the relationship will contain the foreign key pointing to the correlated primary key attribute.

For a foreign key to be created, a primary key must first be properly defined. Once defined, this primary key can be referenced by the foreign key. To quickly summarize the use of primary keys, keep the following in mind:

- Primary keys consist of sets of attributes whose values uniquely identify the rows in an entity.
- Primary keys give an ID to a row. They make the row unique throughout the entity. This means that rows can easily be located by this identifier.

- Primary keys can be used only for columns or attributes that don't allow empty entries. Allowing empty values would mean that a row would not be uniquely identified.
- The attribute chosen to hold a primary key must have values unique throughout the entity.

Foreign keys help in the relational process between two entities. When a primary key is created on a parent entity, it is connected to another entity by linking to the other entity's foreign key. For example, in an invoice situation, there are usually two entities: one for invoice general information and the other for invoice details. The invoice details would contain a hook on to the invoice general entity through the use of a foreign key, potentially the invoice number or a surrogate key.

Keys are usually created as part of the table creation process, but they can be added to the table after the initial generation. The syntax for the creation of keys and their association to tables are discussed in Chapter 3.

Before we are ready to draw all relationships into the model, the data model should begin a process of *normalization*. Some of the relationships can no doubt already be seen; however, when normalization standards are applied to the model, more relationships may be found. Only after the model has been fully normalized will you have a complete view of the entity relationships.

Data Normalization

The process of *normalization* is the division of entities in an attempt to provide the most efficient use of data storage. At times, denormalization is planned redundancy that is subsequently performed to improve response time and better use of resources. The process of designing the logical structure of a database is an attempt to provide a degree of normalization combined with aspects of denormalization to produce optimum storage efficiency while still providing acceptable levels of performance and resource utilization. In most instances, data is fully normalized before any aspects of denormalization are considered. In fact, denormalization usually is not approached until the physical model is in development.

Normalization is usually applied in practice from the outset of data modeling. After you're comfortable with all the modeling concepts, you will find yourself implementing normalized structures as a form of good practice. As

with all other good development habits, you must first work through the concepts at a granular level before they begin to become second nature.

Normalization in general refers to how you implement and store data. Normalization is a design process intended to eliminate duplicate data. In a normalized entity, the redundant data is removed and the entity is simplified to its most basic form. This usually leads to a more involved entity structure with more entities. In the same way, the process of database normalization and planned denormalization is the process of simplifying data and data design to achieve maximum performance and simplicity. This denormalization process involves the planned addition of redundant data.

Although both normalization and denormalization are valid, necessary processes, the two achieve opposite goals. They don't by themselves achieve maximum performance and simplicity, though they do strive for a perfect balance between performance (denormalization) and simplicity (normalization). Normalization means no duplicate data.

Data Normalization with Normal Forms

In 1970, Dr. E. F. Codd designed three regulations a relational database adheres to, known as *normal forms*. Today they are known as the first, second, and third normal forms. (Normal forms do exceed three, but the first three are the only ones widely used.) The goal of the initial database design is to simplify the database into the third normal form.

Using normal forms provides these basic advantages:

- No data redundancy contributing to data integrity
- Index columns for faster sorting and searching
- Smaller entities that reduce table locking and data contention
- Query optimization

Although in most cases a data model is taken only to the third normal form, it is worth noting that there are actually five forms. Because development to the fourth and fifth normal forms is not a requirement for the exam, we mentioned those forms here only for completeness.

The rules provided by these normal forms are discussed in the following sections.

First Normal Form

The *first normal form*, or *1NF*, defines the foundation for the relational database system. An attribute that is represented only once, and thus is not repeating, is known as an *atomic value*. Attributes should be atomic, which means that they cannot (or should not) be further broken down, based on the business needs for the use of the attribute. The first normal form defines that all attributes be atomic, which is to say they cannot be decomposed and must be nonrepeating.

In relational database terms, an attribute of an entity shouldn't have more than one definable piece of data or repeating groups. 1NF states that all attributes must be defined in their most singular form, which means that attributes must be decomposed and not further divisible.

A full name should never be used. For example, a field called `customer name` could be divided into `first name` and `last name` and would therefore break the first normal form rule. The first name is a piece of data that is independent from the last name and therefore it should be a separate attribute.

Second Normal Form

The purpose behind the second normal form is to ensure that each attribute belongs in the entity. Any non-key attribute of an entity must depend on the entire primary key, not just a portion of the key. For example, if the primary key of an `orders` entity contained two fields, `customer id` and `product id`, the attribute field `product description` wouldn't belong, because it has no connection to the `customer id`, only the `product id`. This would break the second normal form rule.

Third Normal Form

The third normal form states that a non-key field must not depend on another non-key field. The most obvious example of this rule is in the case of address information. The zip code is dependent on the address area; the city is dependent on the address itself. A large corporation or government agency may choose to have zip code information stored in a separate table and not within the base data to a perfect normalized form. Although in most table designs this situation is denormalized, in a pure normal form a separate entity would be used to provide additional address information such as city and zip code based on the address.

Two other normal forms do exist but aren't commonly implemented. It's entirely possible that by adhering to a third normal form, you may actually accomplish the fourth and fifth forms.

Fourth and Fifth Normal Forms

The fourth normal form dictates that a third normal form has no multivalued dependencies. In other words, every value of an attribute must appear in at least one row with every other value of the other attribute.

The fifth normal form is intended to eliminate joint dependency constraints. This is a theoretical consideration that is thought to have no practical value. If you disregard these forms, the design of the database might be less than perfect, but it should have no loss of functionality.

Normalizing a database is seemingly good but it can hamper performance. In many cases a designer has to consider denormalizing a database. Planned redundancy or denormalization is often brought into the design to provide for better performance or to clarify data.

Denormalization

Purposely adding redundant data and other fields that disobey normal forms is *denormalization*. Denormalizing as a process is more part of the physical design and will also be revisited during the implementation to improve performance. The concept is covered here for continuity and also to show the contrast with data normalization. After you have a logical design completely normalized, rarely will you keep it in that state as you proceed to the physical design of the actual database.

Although normalization gives you a great deal of storage efficiency and might result in increased performance in some situations, there are some drawbacks to a completely normalized database. You should consider the trade-offs in storage efficiency, performance, and maintainability in your final design.

If you go too far with the normalization process, you might actually reverse the effect you're trying to achieve. Although normalization will reduce data redundancy, result in smaller tables with fewer rows, and provide a logical and consistent form, it will also require table joins for the implementation and will not allow for summary, duplicate, or other data that a user might expect to find in a single table. Normalizing a database design too far can decrease performance and make it difficult to alter the underlying table structure, and might make it harder to work with the data.

Denormalization may occur at any number of levels. At the absolute extreme, a database schema can be completely duplicated to a number of servers across the network by implementing replication. This could be warranted if you need to distribute the access to the data across slow network

links or to multiple remote locations. Many advantages are gained through database replication, because the data is more easily available at the locations where it will be used. The drawback of this is increased maintenance of a number of servers. Also, if database replication isn't configured properly, it could monopolize a WAN. In addition, if there are network problems or there is a poor setup, the data might not be synchronized to a level that keeps it up-to-date. Data can be maintained as an exact duplication against a number of servers, but this would require a high-speed network and the configuration of a two-phase commit.

Other, simpler examples of planned denormalization would be to maintain complete address information for customers, suppliers, employees, and so on in the tables with the rest of their general information. This is what most users expect, and it is difficult to maintain a separate address table. There are no defined rules for denormalization, but some definite guidelines will help you understand what level might be appropriate in a given situation.

Data warehousing schemas often use a denormalized approach referred to as a *star* or *snowflake* schema. This schema structure takes advantage of typical decision support queries by using one central "fact" table for the subject area and many dimension tables containing denormalized descriptions of the facts.

There are also several other situations to consider. If a join requires the implementation of more than three tables, denormalization should be considered. In some situations in which the number of columns in a table can grow very large, a denormalized structure would split the table into more easily handled portions and use a one-to-one relationship to connect the information.

The completed structure will have to be modified over time as the live use of the database warrants. Never consider a database design to be perfect or complete. It often takes several years of actual use to determine the best levels of normalization and denormalization to use.

Some forms of replication and data transfer accommodate redundancy by allowing the data to be on two servers simultaneously. If so, one server is usually treated as a read-only server and is used for offloading query processing from the updatable server. This is discussed in depth later in the book and is definitely an exam topic to be prepared for.

Relationships Between Entities

Relationships are the final component in an ER Model, allowing for a logical linkage between one entity and another. A relationship in an ER Model

connects the data elements of two entities that contain information about the same element. The primary entity in a relationship provides some of the data, and other entities provide further related data. A relationship definition states how two entities are connected.

In the modeling process, we attempt to discover which things are related to one another, and how they are related, within the business problem we are modeling. They are usually defined as a link connecting the entities together based on the number of data elements in one entity that are related to one or more elements in the other entity. This is known as the *cardinality of a relationship*.

The cardinality of a relationship is used to define how many elements in one entity match up with elements of another entity. Relationships are usually defined as a numeric link connecting the entities together based on the number of data elements in one entity that are related to one or more full elements in another entity. It can be described as how many of one thing can relate to how many of something else.

Relationships cause a situation known in data modeling as a *dependency*. A dependency is a circumstance in which one entity either can't exist or has little meaning without at least one other entity in the database. When a dependency exists, it becomes a table relationship in the physical database design. There are three basic types of entity dependencies, and these dependencies are based on element cardinality. They are discussed in the points that follow:

- *One-to-one dependency*—A one-to-one dependency is the rarest form, because each record in one entity correlates to exactly one record in the other.
- *One-to-many dependency*—A one-to-many dependency is the most common form of relationship. One record in a primary entity has ties to many records in a secondary entity.
- *Many-to-many dependency*—A many-to-many dependency exists when many records in one entity can relate to many records in another entity.

Relationships are implemented as parent and child entities. In all cases in the ER Model, a key attribute from a child entity is attached to a related key value in a parent. All cardinality of relationships is implemented in this way, meaning that whether you have a one-to-one, one-to-many, or many-to-many relationship, you always maintain integrity by having the key of the child related to a parent.

Identification of Relationships

Setting up the relationships will finalize a draft of the ER Model. This draft will undergo modifications as the database approaches a physical design. Because at this stage of design all the basic elements of data model have been completed using the Entity Relationship approach, we now have a working model allowing us to proceed further into development. A general listing of attributes for each entity and the relationships between these entities is an important springboard to use to progress through the database design to the eventual completed system.

Implementing the relationships and applying normalization principles are often performed as parallel processes because decisions made in one process effect the other process and vice versa. Normalization helps in determining cardinality and the cardinality is a requirement of each relation. The three basic cardinality types, one-to-one, one-to-many, and many-to-many, are partially a result of knowing the business scenario being modeled and partially derived from applying normal forms.

One-To-One Relationship

The one-to-one type of relationship occurs when one row or data element of an entity is associated with only one row or element in the second entity. This type is used mostly when an entity has an extraordinarily large number of attributes so the entity is split in two to make it easier to manage. Also, an extra entity might be desired when developing the physical storage locations of the data. By separating seldom-used data from more frequently used information, you can accommodate faster data retrieval and updates. It is for this reason that these types of relationships are pulled into the model until the physical design of the database has begun.

In modeling a one-to-one relationship, a common key must be present in each of the entities being related. This common key allows for the collective attributes of both entities to be retrieved using a single value. Consider, for example, a product that has many descriptive attributes. Two product entities could be used to separate the different properties. Each entity would use a product number or similar value as a key. This is illustrated in the model segment shown in Figure 2.4.

In the preceding example the `ProductCommon` entity is used to store the attributes that are most readily used, and the `ProductAtypical` entity contains other attributes that, though still needed, are less frequently used. It is much more common to find relationships existing in a one-to-many cardinality.

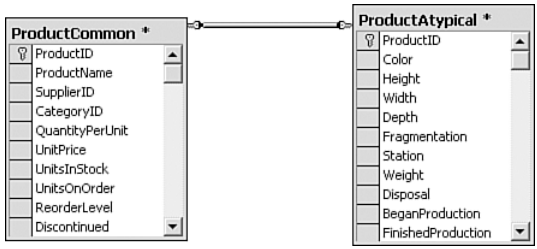

Figure 2.4 Two product entities in a one-to-one relationship.

One-To-Many Relationship

One-to-many relationships exist when a single instance of an entity (the parent entity) relates to many instances of another entity (the child entity). One-to-many dependencies are a natural occurrence in the real world—for example, a customer will have many orders, and a manufactured product could have many components.

One-to-many relationships can be expressed as many-to-one as well, though one-to-many is a common standard. It depends on how the relationship is being viewed.

This relationship is a classic *parent-child dependency*. A foreign key in a child entity will point to the associated primary key of the parent. When this relationship is related, removal of a parent could cause orphaning of the child because of its dependency. There are many examples of this type of relationship in the Northwind order process. The following list represents those seen in the diagram to this point:

- One supplier to many products
- One order to many order details
- One product exists within many order details
- One employee has many orders
- One customer has many orders
- One shipper is used in many orders
- One category will contain many products

Although one-to-many usually establishes "many" as the normal numerical component, you *can* have zero or only one child row. Customers *can* have

zero or one order as well. In fact, cardinality notation allows for this with `0..*`. As we complete more of the information in the business scenario for Northwind, we will see more of these dependencies occur. One-to-many relationships are also a facet of implementing many-to-many relationships, as discussed in the next section.

Many-to-Many Relationship

A modeling differentiation is made in preparing the many-to-many type of relationship. *Many-to-many relationships* exist when many elements of one entity are related to many elements of another. For this reason, many-to-many relationships are implemented a bit differently in a database environment. In itself, this relationship is not solely one entity to another. In the ER model and database design, a third, joining entity is used to complete two one-to-many relations.

This type of relationship is not uncommon in the real world. As stated, a many-to-many relationship is implemented using three entities. The two main entities are connected together using a third entity. The third entity contains keys connected to the other two entities. In our basic data model the `Order Detail` entity is just such an entity. In the Northwind example this entity, however, also exists in its own right and contains additional attributes of its own.

In many models the third entity is created for the purpose of joining two other entities and has no other attributes except for those needed as key values to connect to the original entities. Consider, if you will, an educational scenario in which a teacher instructs several different bodies of students and a student has several different teachers. A third entity, `TeacherStudent`, could be created to connect the main entities.

This new entity that is created is known as an *associate entity* or a *join entity*. Resolving many-to-many relationships involves creating two one-to-many relationships from each of the original entities onto the associative entity. Take the many-to-many relationship between student and teacher. A student can have many teachers while a teacher has many students. A many-to-many relationship will need to be resolved by creating an associative entity and then linking a one-to-many relationship from the `Teacher` and `Student` entities to the `TeacherStudent` entity, as shown in Figure 2.5.

Along with these very standard relationships, you will find in some occurrences that a relationship is made within a single entity. In the case of a relationship in which an entity is related to itself, you have a unique situation, which in modeling terms is called a *unary* or a *self-referencing entity*. In a physical implementation this relationship is implemented through a self-join.

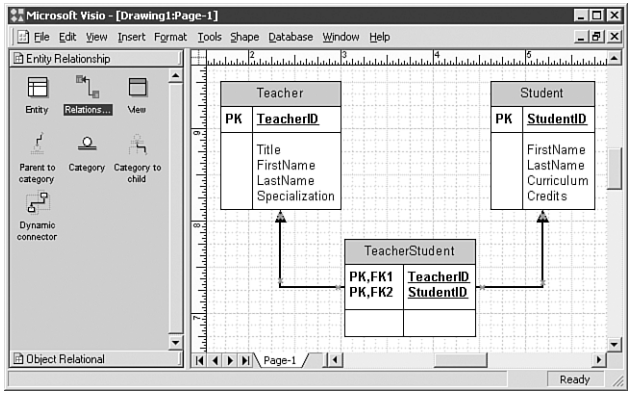

Figure 2.5 A teacher/student many-to-many relationship.

An example of this type of relationship is present within the Northwind system. Within the `Employee` entity, the `ReportsTo` attribute will hold the value of an `Employee Identifier`. This identifier refers to another employee element that is the boss or some other responsible person. The `ReportsTo` element will be a foreign key that refers to the primary key of the `Employee` entity. The modeling of this type of relationship is shown in Figure 2.6.

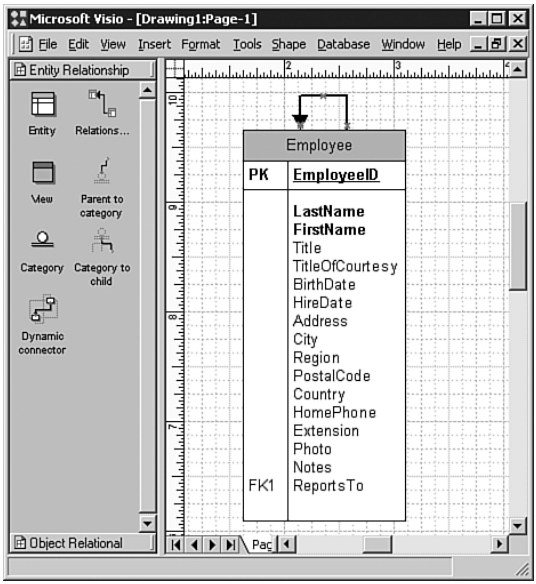

Figure 2.6 A unary relationship.

A unary relationship will constitute a very small percentage of the relationships defined in any given model. These relationships can be defined using a

one-to-one or one-to-many cardinality. In an employee scenario a manager can have a single assistant (one-to-one) or any number of employees can report to the same manager (one-to-many).

Relationships will continue to be added to the model as elements in the system evolve. A potential model for the Northwind Trader order process can now be assembled and is provided in Figure 2.7.

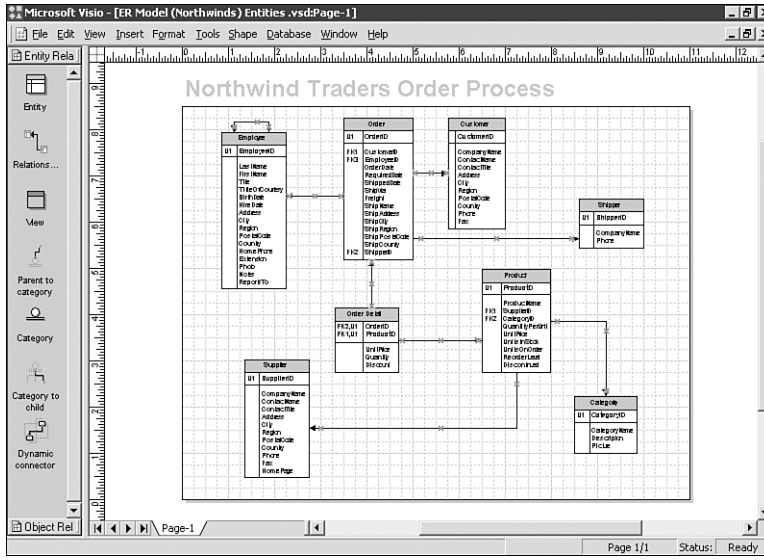

Figure 2.7 The completed Northwind model.

Exam Prep Questions

1. You are a database developer for Northwind Traders. A system that you are developing for your company's SQL Server 2000 will store an online transaction-processing database. Many of the entities are expected to have a very large number of data elements, and these elements will contain a large number of attributes. You want to develop a model for optimal performance. What should you do? (Choose one.)

   ❑ A. Develop a fully normalized structure to minimize the number of joins used to process data.

   ❑ B. Develop a fully normalized structure and then split entities in half, placing the same number of attributes into each entity. Create one-to-one relationships between the two entities.

   ❑ C. Develop a fully normalized structure and then split entities in half based on frequently and infrequently used attributes. Use appropriate relationships to connect the entities.

   ❑ D. Develop a fully normalized structure and then split entities in logical divisions based on commonalities within data element sets to minimize the number of records in each entity.

   ❑ E. Develop a denormalized structure that limits the number of attributes and records in any entity. Create many entities that will have smaller content and apply appropriate relationships.

   Answer: C. Answer A is incorrect because a fully normalized structure will not usually provide for optimum performance. B is not the best approach to provide for performance because there is no measure for usage of attributes or other reasoning for the entity divisions. D, although minimizing the number of rows, does nothing to limit the number of attributes. E is taking the approach too far and will end up with more entities than desired, which will also detract from performance. C is the best approach because it addresses the entity size and performance issues and does so based on sound reasoning.

2. You are a database developer for Northwind Traders. The company is planning to put in place a training facility for employee enrichment purposes. The single room will be scheduled based on the three shifts currently worked by the employees. A senior employee on that shift will teach each course. The entity design, which will also utilize the **Employee** entity, has been roughly sketched and will contain the following:

```
Schedule Entity
. ScheduleID
. CourseID
. EmployeeID
. CourseTime
Course Entity
. CourseID
. CourseTitle
. Description
```

```
. InstructorLastName
. InstructorFirstName
. InstructorTitle
```

You want to promote quick response times for queries and minimize redundant data. What should you do? (Choose one.)

- ❑ A. Create a new table named **Instructors**. Include **InstructorID**, **InstructorFirstName**, **InstructorLastName**, and **InstructorTitle** attributes. Remove these elements from the **Course** entity and replace them with an **InstructorID** attribute.
- ❑ B. Move all the columns from the **Course** entity and place them in the **Schedule** entity, creating just a single entity.
- ❑ C. Remove the **InstructorFirstName**, **InstructorLastName**, and **InstructorTitle** attributes from the **Course** entity. Replace them with an **EmployeeID** attribute.
- ❑ D. Remove the **CourseTime** attribute from the **Schedule** entity and place it into the **Course** entity.

Answer: C. Answer A would be an appropriate answer if the instructors for the courses were external to the company. Because these instructors are internal, the personal information can be drawn from the `Employee` entity, making C a better choice. Making a singular entity as suggested in B would provide far too much redundant storage of data. D is incorrect because the `CourseTime` attribute is a function of the `Schedule` not of the `Course`.

3. You are designing a database model for Northwind Traders that will be used in a customer order process. Customers will be able to order multiple products each time they place an order. You review the model to date, shown here:

```
Customer
.CustomerID
.OrderID
.CompanyName
.ContactName
.Address
.City
.Region
.PostalCode
Order
.OrderID
.ProductID
.OrderDate
.Quantity
.Discount
Product
.ProductID
.Description
.UnitPrice
```

You want good performance while removing redundant data. What should you do? (Each correct answer presents part of a correct solution; choose three.)

- ❑ A. Create a new entity named **OrderDetail**. Add **OrderID**, **ProductID**, **Quantity**, and **Discount** attributes to this entity.
- ❑ B. Ensure that a composite primary key on the **OrderID** and **ProductID** attributes is defined on the **Orders** entity.
- ❑ C. Remove the **ProductID** and **Quantity** attributes from the **Order** entity.
- ❑ D. Decompose the **ContactName** attribute of the **Customer** entity to provide for **FirstName** and **LastName** attributes.
- ❑ E. Move the **UnitPrice** attribute from the **Product** entity to the **Order** entity.
- ❑ F. Remove the **OrderID** attribute from the **Customer** entity and place a **CustomerID** attribute into the **Order** entity.

Answer: A, C, and F. Both A and C are part of the same principle in data modeling and remove the redundant storage of `Discount` information. Because a customer can make many orders, the relationship needs to be made such that an `Order` refers to a `Customer` and not the other way around. The `UnitPrice` attribute is a property of a `Product` and for that reason should stay in that entity. Though the `ContactName` could conceivably be decomposed, there is nothing in the problem statement that would indicate this as a requirement.

4. You are a database consultant for Northwind Traders and you have been hired to develop a database design. This design will be used to develop a database system to be used by a brick-and-mortar store. The information to be maintained in the database will track product categories and suppliers. You create an entity named `Product` that contains the following:

```
Product
.ProductID
.CategoryID
.SupplierID
.QuantityPerUnit
.UnitPrice
.UnitsInStock
.UnitsOnOrder
.ReorderLevel
.Discontinued
```

You must ensure that each product has a valid value for the `Category` and `Supplier` attributes. What should you do? (Choose one.)

- ❑ A. Define the **Product** entity to have a compound primary key that uses the **ProductID**, **CategoryID**, and **SupplierID** attributes.
- ❑ B. Create two relationships in which the **SupplierID** and **CategoryID** attributes each refer to other kernel entities.

- ❑ C. Create a **CategorySupplier** entity and relate the **Product** table to this entity using both the **CategoryID** and the **SupplierID**.
- ❑ D. Remove the **CategoryID** and **SupplierID** attributes from this entity and move them to a more valid kernel entity.

Answer: B. The `CategoryID` and `SupplierID` attributes represent foreign keys that will refer to primary keys within a kernel entity. They are in the correct entity for this purpose and should be referencing the `Category` and `Supplier` entities, respectively.

5. You are designing a portion of the database model that will be used by Northwind Traders for its order process. A quick sketch of the model has been made and is shown here:

```
Product
.ProductID
.Description
.QuantityPerUnit
.UnitsInStock
.Unitprice
.SupplierName
OrderDetail
.OrderID
.ProductID
.CustomerID
.Quantity
.Discount
Order
.OrderID
.OrderDate
.Freight
Customer
.CustomerID
.CompanyName
.ContactName
.Address
.City
.Region
.Phone
.Fax
```

You want to obtain speed and efficiency within the model. What changes should be made? (Choose one.)

- ❑ A. Decompose the **ContactName** attribute so that there are **FirstName** and **LastName** attributes.
- ❑ B. Remove the **SupplierName** attribute from the Product entity and place it into the Order entity.
- ❑ C. Remove the **ProductID** from the **OrderDetail** entity and place it into the **Order** entity.
- ❑ D. Remove the **CustomerID** attribute from the **OrderDetail** entity and place it into the **Order** entity.
- ❑ E. Remove the **Quantity** attribute from the **OrderDetail** entity. Add a **Quantity** column to the **Order** entity.

Answer: D. The `CustomerID` present within the `OrderDetail` entity would be repeated several times per `Order` when it is needed only once. It is therefore more appropriate for the `CustomerID` to be in the `Order` entity.

6. You are a database developer for Northwind Traders. The company is planning a major expansion and desires to begin tracking sales information on a regional basis. Employees of the company will be assigned to a region and are permitted to perform sales only within their designated area. To accommodate this facet of the `Order` process, a rough sketch has been created of two entities that are to be used. These two entities are illustrated here:

```
RegionSale
.RegionSaleID
.OrderID
.RegionID
Region
.RegionID
.RegionTitle
.EmployeeID
```

You would like the new entities to exist within the system as already defined. You would also like to have the system operate quickly with as little redundant information as possible. You would also like key usage to remain consistent with the rest of the system. What should you do? (Select two answers; each answer represents a part of the correct solution.)

- ❑ A. Create only a single entity for the process, combining the attributes from the two sketched entities.
- ❑ B. Create a third new entity, **RegionEmployee**, to connect the **Region** entity to the **Employee** entity.
- ❑ C. Remove the **EmployeeID** attribute and add a **RegionID** attribute to the **Employee** entity.
- ❑ D. Remove the **RegionSaleID** attribute from the **RegionSale** entity.
- ❑ E. Move the **EmployeeID** from the **Region** entity to the **RegionSale** entity.
- ❑ F. Remove the **OrderID** attribute from the **RegionSale** entity.

Answer: C and D. To remain consistent with the other many-to-many relationships in the system, the `RegionSaleID` should be removed and a compound primary key should be based on the `OrderID` and the `RegionID`. The employee should have an attribute for region and not vice versa.

7. You are a database developer for Northwind Traders. You are creating a database model that includes an entity named `Order`. The `Order` entity contains attributes as indicated in the following sketch:

```
Order
.OrderDate
.RequiredDate
.ShipDate
.Freight
```

Employees take orders from the customers and receive a commission on each fulfilled order. Orders can be taken only from the listing of existing customers. Shippers can be selected only from a set of existing shippers. Which additional attributes should be included to complete the entity design? (Choose one.)

- ❑ A. **OrderID**, **CustomerID**, **ShipperID**
- ❑ B. **OrderID**, **CustomerID**, **ShipperID**, **EmployeeID**
- ❑ C. **OrderID**, **ShipperID**, **EmployeeID**
- ❑ D. **OrderID**, **CustomerID**, **EmployeeID**

Answer: B. The `Order` entity as defined by the order process relates to the `Customer`, `Shipper`, and `Employee` entities and should for that reason have foreign keys for each of those kernel entities.

8. You are a database developer for Northwind Traders. The company heads would like to track customer demographics so that they can target advertising budgets and promotions. It is desired to have all budgets based on the past purchases of existing customers. The idea is to target buying patterns by one or more demographics. The demographics to be tracked are the following:

```
gender
age
postal code
region
```

To implement this, area management has sketched the following entities:

```
CustomerDemo
.DemographicID
.DemographicDescription
CustCustomerDemo
.CustomerID
.DemographicID
```

What should you do? (Choose one.)

- ❑ A. Leave the entities as they are to represent an appropriate many-to-many relationship.
- ❑ B. Combine the entities to form one singular entity.
- ❑ C. Add additional attributes to the **CustCustomerDemo** entity.
- ❑ D. Add additional attributes to the **CustomerDemo** entity.

Answer: A. This is a proper many-to-many relationship in which each customer can fit into many demographic categories and any demographic can apply to a number of customers.

9. You are a database developer for Northwind Traders. You are designing a entity to record information about potential new products. A rough sketch of the entity is shown here:

```
TestProduct
.TestProductID
.CategoryID
.SupplierName
.SupplierPhone
.Rating
```

You would like the new entity to be consistent with the remaining system while still storing data in an efficient manner. What should you do? (Choose one.)

- ❑ A. Relate the **TestProduct** entity to the **Product** entity.
- ❑ B. Define a compound primary key that uses both the **TestProductID** attribute and the **CategoryID** attribute.
- ❑ C. Ensure that the **TestProductID** is unique from an existing **ProductID**.
- ❑ D. Replace the **SupplierName** and **SupplierPhone** attributes with a **SupplierID**.

Answer: D. There is already a `Supplier` entity in the system that could easily be used in a relationship with the newly defined `TestProduct` entity. Placement of the `SupplierName` and `SupplierPhone` attributes into this new entity is therefore redundant.

10. As part of the preparation for the database model for Northwind Traders, you have sketched out a set of entities. The sketch as it stands is shown here:

```
Order
.OrderID
.CustomerID
.EmployeeID
.OrderDate
.RequiredDate
.ShippingDate
.Shipvia
.Freight
.Shipname
.ShipAddress
.ShipCity
.ShipRegion
.ShipPostalCode
.ShipCountry
.ShipperID
OrderDetail
.OrderID
.ProductID
.UnitPrice
.Quantity
Product
.ProductID
.ProductName
.SupplierID
.CategoryID
```

```
.QuantityPerUnit
.UnitPrice
.UnitsInStock
.UnitsOnOrder
.ReorderLevel
.Discontinued
Supplier
.SupplierID
.CompanyName
.ContactName
.ContactTitle
.Address
.City
.Region
.PostalCode
.Country
.Phone
.Fax
.HomePage
```

You are now setting up the relationships for the entities. How should these be applied? (Each correct answer represents part of the solution; choose three.)

- ❑ A. Create a one-to-many relationship on the **Product** entity that references the **OrderDetail** entity.
- ❑ B. Create a many-to-one relationship on the **Product** entity that references the **OrderDetail** entity.
- ❑ C. Create a one-to-many relationship on the **Product** entity that references the **Supplier** entity.
- ❑ D. Create a many-to-one relationship on the **Product** entity that references the **Supplier** entity.
- ❑ E. Create a one-to-many relationship on the **Order** entity that references the **OrderDetail** entity.
- ❑ F. Create a many-to-one relationship on the **Order** entity that references the **OrderDetail** entity.

Answer: A, D, and E. There will be many `OrderDetail` elements for each `Order`, many products to a supplier, and many `OrderDetail` elements that refer to any product.