# ERwin

## Methods Guide

***ERwin Version 3.0***

***Methods Guide***

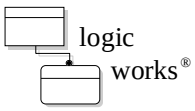
Logic Works, Inc.

University Square at Princeton

111 Campus Drive

Princeton, NJ 08540








Printed in the United States of America.

# Contents

# Preface

Welcome to data modeling with ERwin. If you have never seen a model before, the ***ERwin Methods Guide*** will help you understand what a model is, and what it is good for. If you already have some experience with data and data models, you know how useful they can be in understanding the requirements of your business. A model can help if you are designing new information systems or are maintaining and modifying existing ones.

Data modeling is not something that can be covered in a lot of detail in a short document like this one. But by the time you have read it, you will understand enough, even if you are just a beginner, to put ERwin's *methods* to work for you. Overall, the ***ERwin Methods Guide*** has the following purposes:

- To provide a basic level of understanding of the data modeling method used by ERwin that is sufficient to do real database design.
- To introduce some of the descriptive power and richness of the IDEF1X and IE modeling languages supported by ERwin and to provide a foundation for future learning.
- To provide additional information that lets you to better understand ERwin's modeling features.

This document covers the *methods* of data modeling supported by ERwin, which include:

- IDEF1X. The IDEF1X method was developed by the U.S. Air Force. It is now used in various governmental agencies, in the aerospace and financial industry, and in a wide variety of major corporations.
- IE (Information Engineering). The IE method was developed by James Martin, Clive Finkelstein, and other IE authorities and is widely deployed in a variety of industries.

Both methods are suited to environments where large scale, rigorous, enterprise-wide data modeling is essential.

## Intended Audience

This manual is intended for:

- Novice database designers and data modelers — as a primer on data modeling, and as a guide to using the ERwin methods.
- Experienced data modelers and applications developers — as a guide to IDEF1X and IE data modeling in ERwin.
- Experienced IDEF1X or IE users — as a guide to the features of IDEF1X and/or IE supported by ERwin, and the mapping between these methods.

## About this Guide

This document contains seven chapters, an appendix, a glossary, and an index:

- Chapter 1 describes data modeling, provides a sample methodology for creating a data model, and the benefits of modeling in ERwin.
- Chapter 2 describes the creation of an entity-relationship diagram (ERD) and explains both the process and validation of the model. This chapter also introduces the ideas of entities, attributes, and relationships.
- Chapter 3 explains the concept of keys, including candidate keys, primary and alternate keys, inversion entries, migration of foreign keys, and the use of rolenames.
- Chapter 4 explains the importance of creating accurate names and definitions for entities, attributes, and rolenames in the logical model.
- Chapter 5 provides additional information about entity relationships, including relationship cardinality, referential integrity. This chapter also describes additional relationship types, such as many-to-many, n-ary, recursive, and subtype relationships.
- Chapter 6 defines normalization and the six normal forms of database design. This chapter also provides solutions to a number of common design problems, and describes ERwin's support for normalization.
- Chapter 7 describes physical model constructs and the creation of the physical model from a logical model in ERwin.
- Appendix A describes the types of dependent entities, including characteristic, associative, designative, and subtype entities, and their use in a logical model.

## Typographical Conventions

The ERwin Methods Guide uses some special typographic conventions to identify ERwin user interface controls and key terms that appear in the text.

| Text Item | Convention | Example |
|---|---|---|
| New and important terms | Bold italics | ***normalization*** |
| Entity Name | All uppercase; followed by the word "entity" in lower case | MOVIE COPY entity |
| Attribute Name | All lowercase in quotation marks; hyphen replaces embedded space(s). | "movie-name" |
| Column Name | All lowercase. | movie_name |
| Table Name | All uppercase. | MOVIE_COPY |
| Verb Phrase | All lowercase in angle brackets. | <is available for rental as> |

## Related Documentation

The ERwin documentation set includes the following print and online manuals:

- ***ERwin Online Help***
- ***ERwin Methods Guide (this guide)***
- ***ERwin Reference Guide***
- ***ERwin Features Guide***
- ***ERwin Workgroup Modeling Guide***
- ***Logic Works RPTwin User's Guide***

# Information Systems, Databases, and Models

## What's In This Chapter?

Information systems can have numerous benefits to corporations, from automating tasks that were previously performed manually, to uncovering information and relationships that were previously unknown or undefined. In short, the benefits of information systems can be boiled down to a few key words: faster, better, and more.

However, to realize the benefits of information systems, you must be able to develop them in a timely and cost effective manner, so that they meet real business needs and are modifiable and maintainable with minimum expense. Achieving these goals is a major challenge -- a poorly designed system will end up costing more money and time than it saves.

The most important tool in reducing the cost of managing and retrieving information has become the relational database management system or RDBMS. An RDBMS provides a reliable and convenient means of storing, retrieving, and updating data.

Equally important is a method that reduces the cost of designing and managing relational databases. The most important and widely used method is called ***data modeling***.

### Chapter Contents

## What is Data Modeling?

Data modeling is the process of describing information structures and capturing business rules in order to specify information system requirements. A data model represents a balance between the specific needs of a particular RDBMS implementation project, and the general needs of the business area that requires it.

Structured system development approaches in general, and data-centered design approaches specifically, invest heavily in front end planning and requirements analysis activities. Many of these "top-down" design approaches use ERwin data modeling as a method for identifying and documenting that portion of system requirements relating to data. Process models (e.g., data flow diagram sets, distribution models, event/state models) can be created in Logic Works BPwin and other tools to document processing requirements. Different levels of these models are used during different development phases.

When created with the full participation of business and systems professionals, the data model can provide many benefits. These benefits generally fall into two classes — those primarily associated with the model (the product of the effort) and those associated with the process of creating the model (the effort).

Examples of product benefits:

- A data model is implementation-independent, so it does not require that the implementation is in any particular database or programming language.
- A data model is an unambiguous specification of what is wanted.
- The model is business user-driven. The content and structure of the model is controlled by the business client rather than the system developer. The emphasis is on requirements rather than constraints or solutions.
- The terms used in the model are stated in the language of the business, not that of the system development organization.
- The model provides a context to focus discussions on what is important to the business.

Examples of process benefits:

- During early project phases, model development sessions bring together individuals from many parts of the business, and provide a structured forum in which business needs and policies are discussed. During these sessions, it is often the case that the business staff, for the first time, meet

others in different parts of the organization who are concerned with the same needs.

- Sessions lead to development of a common business language with consistent and precise definitions of terms used. Communication among participants is greatly increased.
- Early phase sessions provide a mechanism for exchanging large amounts of information among business participants, and transferring much business knowledge to the system developers. Later phase sessions continue that transfer of knowledge to the staff who will implement the solution.
- Session participants are generally able to better see how their activities fit into a larger context. And parts of the project can be seen in the context of the whole. The emphasis is on “cooperation” rather than “separation.” Over time, this can lead to a shift in values, and the reinforcement of a cooperative philosophy.
- Sessions foster consensus and build teams.

Design of the data structures to support a business area is only one part of developing a system. The analysis of processes (function) is equally important. Function models describe “how” something is done. They can be presented as hierarchical decomposition charts, data flow diagrams, HIPO diagrams, etc. You will find, in practice, that it is important to develop both your function and data models at the same time. Discussion of the functions to be performed by the system uncovers the data requirements. Discussion of the data normally uncovers additional function requirements. Function and data are the two sides of the system development coin.

ERwin provides direct support for process modeling and can work well with many techniques. For example, Logic Works also provides a function modeling tool, BPwin, that supports IDEF0, IDEF3 work flow, and data flow diagram methods and can be used in conjunction with ERwin to complete an analysis of process during a data modeling project.

## Data Modeling Sessions

Creating a data model involves not only construction of the model, but also numerous fact-finding sessions that uncover the data and processes used by a business. Running good sessions, like running good meetings of any kind, depends on a lot of preparation, and "real-time" facilitation techniques. In general, modeling sessions should include the right mix of business and technical experts and should be facilitated. This means that they are scheduled well in advance, carefully planned to cover sets of focused material, and orchestrated in such a way that desired results are achieved.

When possible, it is highly recommended that modeling of function and data be done at the same time. This is because functional models tend to validate a data model and uncover new data requirements. This approach also ensures that the data model supports function requirements. To create both a function and data model in a single modeling session, it is important to include not only a data modeler, but also a process modeler who is responsible for capturing the functions being explored.

### Session Roles

Formal, guided sessions, with defined roles for participants and agreed upon procedures and rules, are a must. The following roles seem to work well:

- The facilitator is the session guide. This person is responsible for arranging the meetings and facilities, providing follow-up documentation, and intervening during sessions, as necessary, to keep them on track and control the scope of the session.
- The data modeler is responsible for leading the group through the process of developing the model and validating it. The modeler develops the model, in real-time if possible, in front of the group by asking pertinent questions that bring out the important detail,and recording the resulting structure for all to see. It is often possible (although somewhat difficult) for the same individual to play both facilitator and data modeler roles.
- The data analyst(s) functions as the scribe for the session, and records the definitions of all entities and attributes that make up the model. They can also begin to "package" entities and attributes into subject areas, manageable and meaningful subsets of the complete data model, based on information from the business experts.

- The subject matter experts provide the business information needed to construct the model. They are “business” not “systems” people. It is their business that is being analyzed.
- The manager, either from the “systems” or “business” community, participates in the sessions in his or her assigned role (facilitator, subject matter expert, etc.) but has the additional responsibility of making decisions as needed to keep the process moving. The manager has the responsibility of “breaking ties” but only when absolutely necessary.

## Sample IDEF1X Modeling Methodology

ERwin has been developed to support the IDEF1X and IE modeling standards, but is otherwise free from restrictions on any methodology. The use of various levels of models within the IDEF1X method, however, can be very helpful in developing a system. General model levels are outlined in the IDEF1X standard, and are presented below. In practice, you may find it useful to expand or contract the number of levels to fit individual situations.

The model levels generally trend from a very wide, but not too detailed view of the major entities that are important to a business, down to a level of precision required to represent the database design in terms understandable by a particular DBMS. At their very lowest level of detail, the models are said to be technology dependent, e.g., a model for an IMS database will look very different from a model for a DB2 database. At higher levels, the models are technology independent, and may even represent information which is not stored in any automated system.

The modeling levels presented below are well suited to a top-down system development life cycle approach, in which successive levels of detail are created during each project phase.

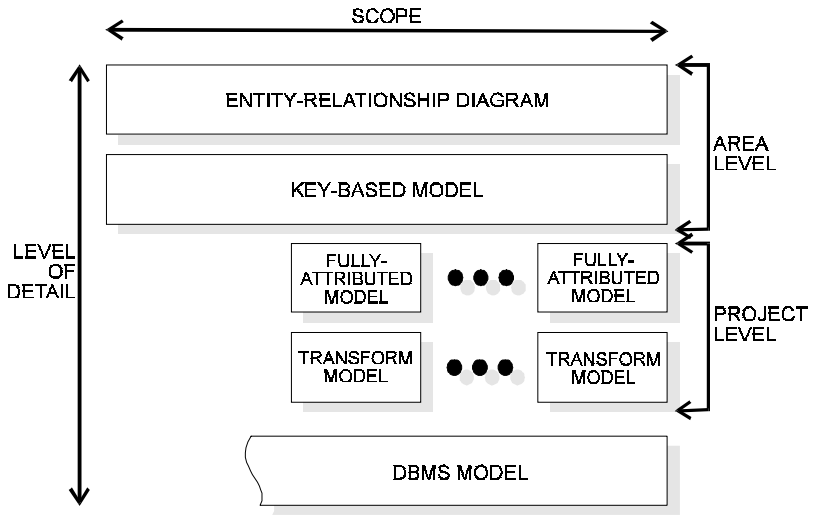
The highest level models come in two forms. The Entity Relationship Diagram (ERD) identifies major business entities and their relationships. The Key-Based (KB) Model sets the scope of the business information requirement (all entities are included) and begins to expose the detail.

The lower level models also come in two forms. The Fully Attributed (FA) Model is a third normal form model which contains all of the detail for a particular implementation effort. The Transformation Model (TM) represents a transformation of the relational model into a structure which is appropriate to the DBMS chosen for implementation.

The transformation model, in most cases, is no longer in third normal form. The structures have been optimized based on the capabilities of the DBMS, the data volumes, and the expected access patterns and rates against the data. In a sense, this is a picture of the eventual physical database design.

The database design is contained in the DBMS Model for the system. Depending on the level of integration of a business' information systems, the DBMS Model may be a project level model, or an area level model for the entire integrated system.

These levels are presented in the figure below. Notice that the DBMS Model can be either at an “Area” scope, or a “Project” scope. It would not be uncommon to have single ERD and KB models for a business, and multiple DBMS Models, one for each implementation environment, and then another set within that environment for “projects” which do not share databases. In an ideal situation, there are a set of “Area” scope DBMS Models, one for each environment, with complete data sharing across all projects in that environment.




*IDEF1X Database Design Levels*

## Logical Models

There are three levels of logical models that are used to capture business information requirements: the Entity Relationship Diagram (ERD), the Key-Based (KB) Model, and the Fully Attributed(FA) model. The ERD and KB models are also called “area data models” because they often cover a broad business area that is larger than the business chooses to address with a single automation project. In contrast, the FA model is a “project data model” because it typically describes a portion of an overall data structure intended for support by a single automation effort.

### The Entity Relationship Diagram

The Entity Relationship Diagram is a high level data model that shows the major entities and relationships which support a wide business area.

The objective of the entity relationship diagram is to provide a view of business information requirements sufficient to satisfy the need for broad planning for development of its information system. These models are not very detailed (only major entities are included,) and there is not much detail, if any, on attributes. Many-to-many (non-specific) relationships are allowed, and keys are generally not included. This is primarily a presentation or discussion model.

### The Key-Based Model

A Key-Based Model describes the major data structures which support a wide business area. All entities and primary keys are included, along with sample attributes.

The objective of the key-based model is to provide a wide business view of data structures and keys needed to support the area. This model provides a context in which detailed implementation level models can be constructed. The model covers the same scope as the Area ERD, but exposes more of the detail.

### The Fully-Attributed (FA) Model

A Fully Attributed Model is a third normal form data model that includes all entities, attributes and relationships needed by a single project. The model includes entity instance volumes, access paths and rates, and expected transaction access patterns against the data structure.

## Physical Models

There are also two levels of physical models for an implementation project: the Transformation Model and the DBMS Model. The physical models capture all of the information that systems developers need to understand and implement a logical model as a database system. The Transformation Model is also a "project data model" that describes a portion of an overall data structure intended for support by a single automation effort. ERwin supports individual projects within a business area, allowing the modeler to separate an larger area model into submodels, called subject areas. Subject areas can be developed, reported on, and generated to the database in isolation from the area model and other subject areas in the model.

### The Transformation Model

The objectives of the transformation model are to provide the Database Administrator (DBA) with sufficient information to create an efficient physical database, to provide a context for the definition and recording in the data dictionary of the data elements and records that form the database, and to assist the application team in choosing a physical structure for the programs that will access the data.

When deemed appropriate for the development effort, the model can also provide the basis for comparing the physical database design against the original business information requirements — to demonstrate that the physical database design adequately supports those requirements, to document physical design choices and their implications (e.g., what is satisfied, and what is not), and to identify database extensibility capabilities and constraints.

### The DBMS Model

The transformation model directly translates into a DBMS model, which captures the physical database object definitions in the RDBMS schema or database catalog. ERwin directly supports this model with its schema generation function. Primary keys become unique indices. Alternate keys and inversion entries also may become indices. Cardinality can be enforced either through the referential integrity capabilities of the DBMS, application logic, or "after the fact" detection and repair of violations.

## Benefits of Modeling in ERwin

Regardless of the type of DBMS you use or which types of data models you wish to develop, modeling your database in ERwin has many benefits. The most obvious benefit is system documentation that can be used by database and application development staff to define system requirements and to communicate among themselves and to end-users.

A second benefit is to provide a clear picture of referential integrity constraints. Maintaining referential integrity is essential in the relational model where relationships are encoded implicitly.

A third benefit is the provision of a “logical” RDBMS-independent picture of your database that can be used by automated tools to generate RDBMS-specific information. Thus, you can use a single ERwin diagram to generates DB2 table schemas, as well as schemas for other relational DBMSs.

One of the primary benefits of data modeling with ERwin is the ease with which you will be able to produce a diagram summarizing the results of your data modeling efforts, and generate a database schema from that model.

# 2 Constructing a Logical Model

## What's In This Chapter?

The first step in constructing a logical model is developing the Entity Relationship Diagram (ERD), a high level data model of a wide business area. An entity-relationship diagram is made up of three main building blocks — entities, attributes, and relationships. If we view a diagram as a graphical language for expressing statements about your business, entities are the nouns, attributes are the adjectives or modifiers, and relationships are the verbs. Building a data model with ERwin is simply a matter of finding the right collection of nouns, verbs, and adjectives and putting them all together.

The objective of the ERD is to provide a view of business information requirements sufficient to satisfy the need for broad planning for development of its information system. These models are not very detailed (only major entities are included,) and there is not much detail, if any, on attributes. Many-to-many (non-specific) relationships are allowed, and keys are generally not included. This is primarily a presentation or discussion model.

ERDs are also divided into ***subject areas***, which are used to define "business views" or specific areas of interest to individual business functions. Subject areas help reduct larger models into smaller packages more manageable subsets of entities, that can be more easily defined and maintained.

There are many methods available for developing the ERD. These range from formal modeling sessions (described in the previous chapter) to individual interviews with business managers who have responsibility for wide areas.

This chapter introduces the data modeling method used by ERwin and provides a brief overview of its richness and power for describing the information structures of your business.

### Chapter Contents

## The Entity-Relationship Diagram

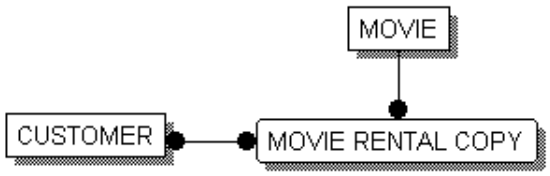
If you are familiar with a relational database structure, you know that the most fundamental component of a relational database is the ***table***. Tables are used to organize and store information. A table is organized in ***columns*** and ***rows*** of data. Each row contains a set of facts called an ***instance*** of the table.

In a relational database, all data values must also be atomic — each cell in the table can contain only a single fact. There is also a relationship between the tables in the database. Each relationship is represented in an RDBMS by sharing one or more columns in two tables.

Like the tables and columns that make up a physical model of a relational database, an entity-relationship diagram (and all other logical data models) include equivalent components that let you model the data structures of the business, rather than the database management system. The logical equivalent to a table is an ***entity***, and the logical equivalent to a column is an ***attribute***.

In an ERD, the entity is represented by drawing a box that contains the with the name of the entity. Entity names are always singular — CUSTOMER not CUSTOMERS, MOVIE not MOVIES, COUNTRY not COUNTRIES. By always using singular nouns, you gain the benefit of a consistent naming standard and facilitate "reading" the diagram as a set of declarative statements about entity instances.

The diagram below is one created by a hypothetical video store that needs to track its customers, movies that can be rented or purchased, and rental copies of movies that are in stock in the store.




*Sample Entity-Relationship Diagram*

Relationships between tables are a vital component of a relational database. These relationships are captures using shared key: facts in one table refer to, or are associated with, facts in another table. In an ERD, a relationship is represented by a line drawn between the entities in the model. A relationship between two entities also implies that facts in one entity refer to, or are associated with, facts in another entity.

In the example above, the video store needs to track information CUSTOMERs and MOVIE RENTAL COPYs. The information in these two entities is related, and this relationship can be expressed in a statement: A CUSTOMER rents one or more MOVIE RENTAL COPYs.

## Defining Entities and Attributes

We can define an entity as any person, place, thing, event, or concept about which information is kept. More precisely, we can think of an entity as a set or collection of like individual objects called instances. An instance is a single occurrence of a given entity. Each instance must have an identity distinct from all other instances.

In the previous example, we might say that the CUSTOMER entity represents the set of all of the possible customers of a business. Each instance of the CUSTOMER entity is a customer. You can list information for an entity in a ***sample instance table***, such as the one shown below.

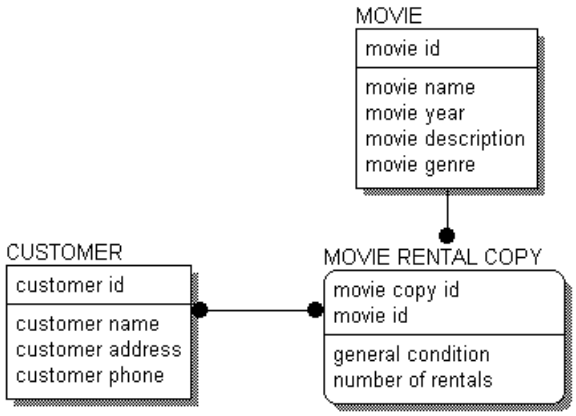
**CUSTOMER**

| customer-id | customer-name | customer-address |
|---|---|---|
| 10001 | Ed Green | Princeton, NJ |
| 10011 | Margaret Henley | New Brunswick, NJ |
| 10012 | Tomas Perez | Berkeley, CA |
| 17886 | Jonathon Walters | New York, NY |
| 10034 | Greg Smith | Princeton, NJ |

*Sample Instance Table for the CUSTOMER Entity*

Each instance represents a set of "facts" about the related entity. In the sample above, each instance of the CUSTOMER entity includes information on the "customer-id," "customer-name," and "customer-address." In a logical model, these properties are called the *attributes* of an entity. Each attribute captures a single piece of information about the entity.

You can include attributes in an ERD to more fully describe the entities in the model, as shown below:




*ERD With Attributes*

## Logical Relationships

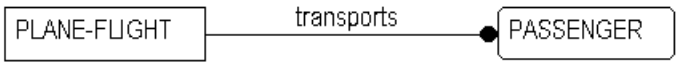
Relationships represent connections, links or associations between entities. They are the "verbs" of a diagram showing how entities relate to each other. Easy-to-understand rules help business professionals validate data constraints, and ultimately identify relationship cardinality.

Here are some examples:

- A TEAM <has> many PLAYERs.
- A PLANE-FLIGHT <transports> many PASSENGERs.
- A DOUBLES-TENNIS-MATCH <requires> exactly 4 PLAYERs.
- A HOUSE <is owned by> one or more OWNERs.
- A SALESPERSON <sells> many PRODUCTs.

In all of these cases, the relationships are chosen so that the connection between the two entities is what is known as ***1-to-many***. This means that one (and only one instance) of the first entity is related or connected to many instances of the second entity. The entity on the "1-end" is called the parent entity, the entity on the "many-end" is called the child entity.

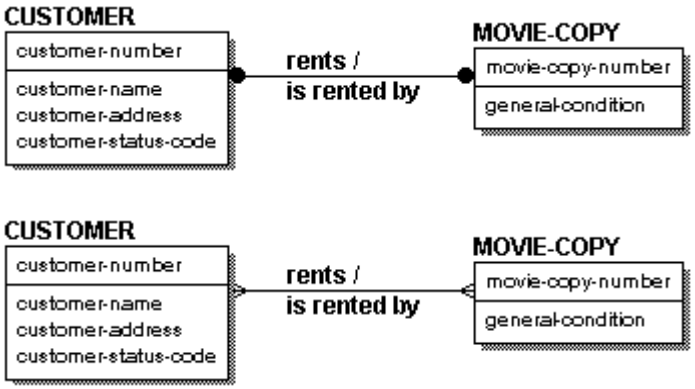
Relationships are displayed as a line connecting two entities, with a dot on one end, and a ***verb phrase*** written along the line. In the previous examples, the verb phrases are the words inside the brackets (e.g., <sells>). Here is a diagram of the relationship between PLANE-FLIGHTs and PASSENGERs on that flight.




*Relationship Example*

## Many-to-Many Relationships

A many-to-many relationship, also called a ***non-specific relationship***, represents a situation where an instance in one entity relates to one or more instances in a second entity, and an instance in the second entity also relates to one or more instances in the first. In the video store example, a many-to-many relationship occurs between a CUSTOMER and a MOVIE COPY. From a conceptual point of view, this many-to-many relationship indicates that "A CUSTOMER <rents> many MOVIE COPYs" and "A MOVIE COPY <is rented by> many CUSTOMERs."




*Example of a Many-to-Many Relationship in IDEF1X (top) and IE (bottom)*

Many-to-many relationships tend to be used in a preliminary stage of diagram development, such as in an entity-relationship diagram (ERD) and are represented in IDEF1X as a solid line with dots on both ends.

Because a many-to-many relationship can hide other business rules or constraints, they should be fully explored at some point in the modeling process. For example, sometimes a many-to-many relationship identified in early modeling stages is mislabeled, and is actually two one-to-many

relationships between related entities. Or, the business must keep additional facts about the many-to-many relationship, such as dates or comments, and the result is that the many-to-many relationship must be replaced by an additional entity to keep these facts. You should ensure that all many-to-many relationships are fully discusssed at later modeling stages to ensure that the relationship is correctly modeled.

## Validating the Design of the Logical Model

If you choose your verb phrases correctly, you should be able to "read" a relationship from the parent to the child using an "active" verb phrase. One of the previous examples read as:

**A PLANE FLIGHT <transports> many PASSENGERs.**

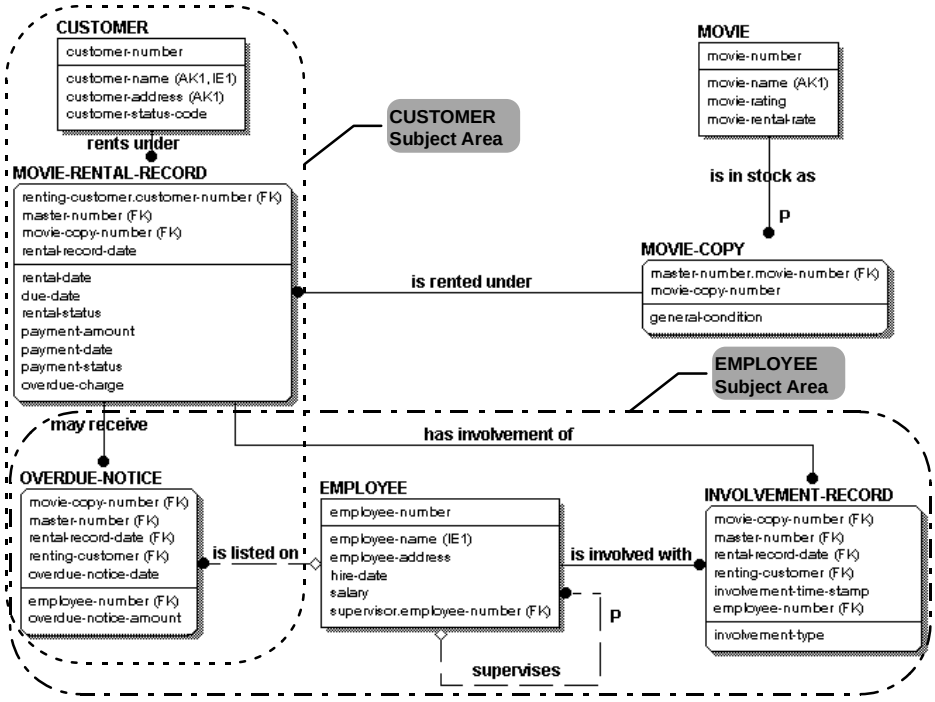
Verb phrases can also be read from the perspective of the child entity. You can often read from the child entity perspective using "passive" verb phrases. For example:

**Many PASSENGERs <are transported by> a PLANE FLIGHT.**

Because a data model exposes many of the business rules that describe the area being modeled, reading the relationships helps you validate that the design of the logical model is correct. Verb phrases provide a brief summary of the business rules embodied by relationships. And although they do not precisely describe the rules, they let the person looking at the model get an initial sense of how the entities are connected. It is a good practice to make sure that each verb phrase in the model results in valid statements. Reading your model back to the business is one of the primary methods of verifying that it has correctly captured the business rules.

## Data Model Example

ERwin includes a model of a database constructed for a hypothetical video store. A copy of the logical model in the MOVIES diagram appears below.




*Data Model for a Video Store*

The data model of the Video Store, along with definitions of the objects presented on it, makes the following assertions:

- A MOVIE is in stock as one or more MOVIE-COPYs. Information recorded about a MOVIE includes its name, a rating, and rental rate. Each MOVIE-COPY has its general condition recorded.
- The store's CUSTOMERs rent the MOVIE-COPYs. A MOVIE-RENTAL-RECORD records the particulars of the rental of a MOVIE-COPY by a CUSTOMER. The same MOVIE-COPY may, over time, be rented to many CUSTOMERs.

- Each MOVIE-RENTAL-RECORD also records a due date for the movie, and a status indicating whether or not it is overdue. Depending on his or her previous relationship with the store, a CUSTOMER is assigned a credit status code which indicates whether the store should accept checks or credit cards for payment, or take only cash.
- The store's EMPLOYEEs are involved with many MOVIE-RENTAL-RECORDs, as specified by an involvement type. There must be at least one EMPLOYEE involved with each record. Since the same EMPLOYEE might be involved with the same rental record several times on the same day, involvements are further distinguished by a time stamp.
- An overdue charge is sometimes collected on a rental of a MOVIE-COPY. OVERDUE-NOTICEs are sometimes needed to remind a CUSTOMER that a tape needs to be returned. An EMPLOYEE is sometimes listed on an OVERDUE-NOTICE.
- The store keeps salary and address information about each of its EMPLOYEEs. It sometimes needs to look up CUSTOMERs, EMPLOYEEs, and MOVIEs by their names, rather than by their "numbers."

This is a relatively small model, but it says a lot about the video rental store. From it, we not only can get an idea of what a database for the business should look like, but we also get a good picture of the business. There are several different types of graphical "objects" in this diagram. The entities, attributes and relationships, along with the other symbols describe our business rules. In the following chapters, you will learn more about what the different graphical objects mean and how to use ERwin to create your own logical and physical data models.

# 3 The Key-Based Model

## What's In This Chapter?

A Key-Based Model (KB) is data model that fully describes all of the major data structures that support a wide business area. The goal of a key-based model is to include all entities and attributes that are of interest to the business.

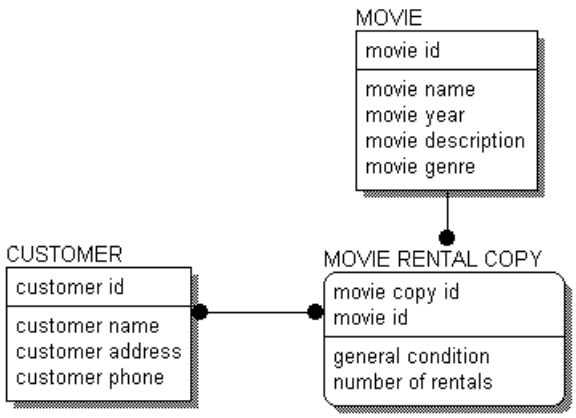
As their name suggests, key-based models also include ***keys***, which are the elements of the data model that are used to identify unique instances within an entity and, when implemented in a physical model, provide easy access to the underlying data.

Basically, the key-based model covers the same scope as the ERD, but exposes more of the detail, including the context in which detailed implementation level models can be constructed.

### Chapter Contents

## Understanding Keys

Lets look at our previous example.




*ERD With Attributes*

Each entity is divided by a horizontal line that separates the attributes into two groups. In fact, this horizontal line divides the attributes into ***keys*** and ***non-keys***. The area above the line is called the ***key area***, and the area below the line is called the data area. The key area of CUSTOMER contains "customer-number," the data area contains "customer-name," "customer-address," and "customer-phone."

The key area contains the ***primary key*** for the entity. The primary key is a set of attributes that the business has chosen to identify unique instances of an entity. The primary key can comprise one or more ***primary key attributes***, as long as the attributes chosen form a unique identifier for each instance in an entity.

Primary key attributes are placed above the line in the key area. As the name suggests, a ***non-key attribute*** is an attribute which has not been chosen as a key. Non-key attributes are placed below the line, in the data area.

Whenever you create an entity in your data model, one of the most important questions you need to ask is: "How can a unique instance be identified?" You must be able to uniquely identify each instance in an entity in order to correctly develop a logical data model. As a reminder, entities in an ERwin model always include a key area so you define key attributes in every entity.

## Selecting a Primary Key

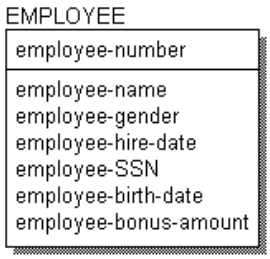
Choosing the primary key of an entity is an important step, and requires some serious consideration. There may be several attributes, or sets of attributes that could be used as primary keys. Attributes or groups of attributes that can be chosen as primary keys are called ***candidate key attributes***. A candidate key must uniquely identify each instance of the entity. Accordingly, no part of the key can be NULL, that is "empty" or "missing." The business user is typically the best person to identify candidate keys, because of their knowledge of the business and business data.

For example, in order to correctly use the EMPLOYEE entity in a data model (and later in a data base), you must be able to uniquely identify instances. In the customer table, you could choose from several potential key attributes including: the employee name, a unique employee number assigned to each instance of EMPLOYEE, or a group of attributes, such as name and birth date.

The rules that you use to select a primary key from the list of all candidate keys are stringent, but can be consistently applied across all types of databases and information. The rules state that the attribute or attribute group must:

- Uniquely identify an instance.
- Never include a NULL value.
- Not change over time. An instance takes its identity from the key. If the key changes, it's a different instance
- Be as short as possible, to facilitate indexing and retrieval. If you need to use a key that is a combination of keys from other entities, make sure that each part of the key adheres to the other rules.

Consider the following example:




*Key selection example*

If you use the rules listed above to find candidate keys for EMPLOYEE, you might compose the following analysis of each attribute:

- Because it is unique for all EMPLOYEEs, “employee-number” is a candidate key.
- “Employee-name” does not look like a good candidate. There may be two John Smiths in the company.
- “Employee-social-security-number” is unique, but every EMPLOYEE may not have one.
- The combination of “employee-name” and “employee-birth-date” might work (unless there are two John Smiths, born on the same date, and both employed by our company). This could be a candidate key.
- Only some EMPLOYEEs of our company are eligible for annual bonuses. Therefore, “employee-bonus-amount” can be expected to be NULL in many cases. As a result, it cannot be part of any candidate key.

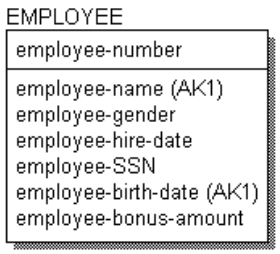
After analysis, there are two candidate keys — one is “employee-number” and the other is the group of attributes containing “employee-name” and “employee-birth-date.” Because it is the shortest, and ensures uniqueness of instances “employee-number” is selected as the primary key.

When choosing the primary key for an entity, modelers often assign a *surrogate key*, an arbitrary number that is assigned to an instance to uniquely identify it within an entity. “Employee-number” is an example of a surrogate key. A surrogate key is often the best choice for a primary key because is short and can be accessed the fastest, and ensures unique identification of each instance. Further, a surrogate key can be automatically generated by the system, so that numbering is sequential and does not include any gaps.

It is acceptable to choose a primary key for the logical model, only to discover that the primary key needed to efficiently access the table in a physical model is different. The key can be changed to suit the needs and requirements of the physical model and database at any point.

## Designating Alternate Key Attributes

Candidate keys not selected as primary keys can be designated as ***alternate keys***, and recorded as such in the model. The symbol (AK*n*), where *n* is a number, is placed after those attributes which form the alternate key. Alternate keys are often used to show different indexes the business will use to access the data. So our logical model for EMPLOYEE appears as follows:

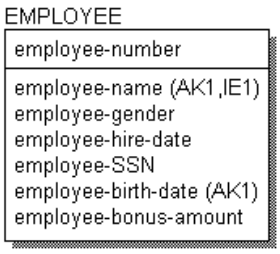



*Alternate Key Example*

## Inversion Entry Attributes

Businesses also need to keep track of attributes that are not unique, but are routinely used to look up information for the entity. These attributes are called ***inversion entries.*** An inversion entry is an attribute or group of attributes that are commonly used to access the entity (as though they are a primary key), but may not result in finding exactly one instance.

For example, the business might want to be able to look up an employee by name, as well as the employee number. Although a search on a name may result in one, two, or more records, it is still a business requirement that employee records can also be accessed using the employee name. When you assign the attribute to an inversion entry, an IE*n* is placed after the "employee-name" attribute, as shown below. There can be several inversion entries for an entity.

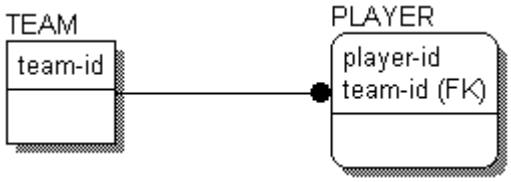



*Inversion Entry Example*

## Relationships and Foreign Key Attributes

Just as a RDBMS captures relationships using shared key values, ERwin also represents relationships using shared keys. Although ERwin certainly can be used to model information that is stored in non-relational data base management systems, in its treatment of keys, ERwin is relational.

Whenever entities in a ERwin diagram are connected by a relationship, the relationship contributes a key (or set of key attributes) to the child entity. These attributes are called the foreign key. ***Foreign key attributes*** are defined as primary key attributes of a parent entity contributed to a child entity across a relationship. The contributed keys are said to ***migrate*** from parent to child. Foreign key attributes are designated in the model by an (FK) following the attribute name. Notice the (FK) next to "parent-key" in the figure below.




*PLAYER Entity With Migrated Foreign Key (FK)*

### Dependent and Independent Entities

As you develop your model, you may discover certain entities that depend on the value of the foreign key attribute for uniqueness. For these entities, the foreign key must be a part of the primary key of the child entity (above the line) in order to uniquely define each entity.

In relational terms, a child entity that depends on the foreign key attribute for uniqueness is called a ***dependent entity***. In the example above, PLAYER is considered a dependent entity because it depends on the TEAM entity for its identification. In IDEF1X, dependent entities are represented as round-cornered boxes.

Dependent entities are further classified as ***existence dependent***, which means the dependent entity cannot exist unless its parent does, and ***identification dependent***, which means that the dependent entity cannot be identified without using the key of the parent. The PLAYER entity is identification dependent, but not existence dependent, because PLAYERs can exist if they are not on a TEAM.

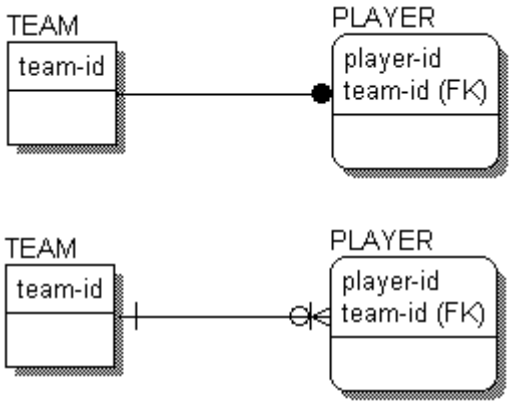
In contrast, there are situations in which an entity is existence dependent on another entity. Consider two entities: ORDER, which the business uses to track customer orders, and LINE ITEM, which tracks individual items in an ORDER. The relationship between these two entities can be expressed as An ORDER <contains> one or more LINE ITEMS. In this case, LINE ITEM is existence dependent on ORDER, because it makes no sense in the business context to track LINE ITEMS unless there is a related ORDER.

Entities that do not depend on any other entity in the model for identification are called ***independent*** entities. In the example above, TEAM is considered an independent entity. In IE and IDEF1X, independent entities are represented as square-cornered boxes.

## Identifying Relationships

In IDEF1X, the concept of dependent and independent entities is enforced by type of the relationship that connects two entities. If you want the foreign key to migrate to the key area of the child entity (and create a dependent entity as a result), you can create an ***identifying*** relationship between the parent and child entities.

Identifying relationships are indicated by a solid line connecting the entities. In IDEF1X, the line includes a dot on the end nearest to the child entity, as shown below. In IE, the line includes “crows feet” at the end of the relationship nearest to the child entity.




*Identifying Relationship in IDEF1X Notation (top) and IE Notation (bottom)*

***Note**: Standard IE notation does not include rounded corners on entities. This is an IDEF1X symbol that is included in IE notation in ERwin to ensure compatibility between methods.*

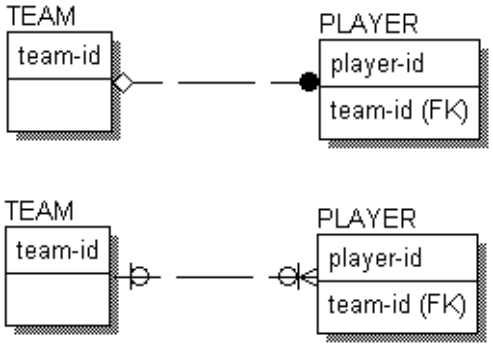
As you saw in the discussion of independent and dependent entities, the business rule that indicates that a relationship is identifying results from an intentional choice to identify the child entity by using the identifier of the parent entity. In our example of MOVIEs and MOVIE-COPYs, we could have chosen to identify the copy by its own unique number. Instead, we decided to use the identifier of the MOVIE and add a second part (copy-number) to tell one copy from another.

***Note**: As you may find, there are advantages to contributing keys to a child entity through identifying relationships in that it tends to make some physical system queries more straightforward, but there are also many disadvantages. Some advanced relational theory suggests that contribution of keys should not occur in this way. Instead, each entity should be identified not only by its own primary key, but also by a logical handle or surrogate key, never to be seen by the user of the system. There is a strong argument for this in theory and those who are interested are urged to review the work of E. F. Codd and C. J. Date in this area.*

## Non-Identifying Relationships

***Non-identifying relationships***, which are unique to the IDEF1X notation, also connect a parent entity to a child entity. Non-identifying relationship are used to show a different migration of the foreign key attribute(s) – migration to the data area of the child entity (below the line).

Non-identifying relationships are indicated by a dashed line connecting the entities. If you connect the TEAM and PLAYER entities in a non-identifying relationship, the model appears as shown below.




*Non-Identifying Relationship in IDEF1X Notation (Top) and IE Notation (bottom)*

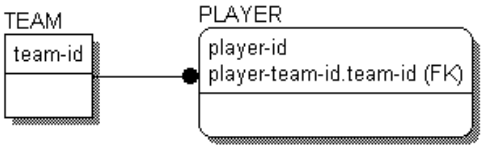
Because the migrated keys in a non-identifying relationship are not part of the primary key of the child, non-identifying relationships do not result in any identification dependency. In this case, PLAYER is considered an independent entity, just like TEAM.

However, the relationship can reflect existence dependency if the business rule for the relationship specifies that the foreign key cannot be NULL ("missing"). If the foreign key must exist, this implies that an instance in the child entity can only exist if an associated parent instance also exists.

***Note**: Identifying and non-identifying relationships are not a feature of the IE method. However, this information is included in your ERwin diagram in the form of a solid or dashed relationship line to ensure compatibility between IE and IDEF1X methods.*

## Rolenames

When foreign keys migrate from the parent entity in a relationship to the child entity, they are serving double-duty in the model in terms of stated business rules. To understand both roles, it is sometimes helpful to rename the migrated key to show the role it plays in the child entity. This name assigned to a foreign key attribute is called a ***rolename***. In effect, a rolename declares a new attribute, whose name is intended to describe the business statement embodied by the relationship that contributes the foreign key.




*Rolename Example*

The foreign key attribute of PLAYER, "player-team-id.team-id," shows us the syntax for defining and displaying a rolename. The first half (before the period) is the rolename. The second half is the original name of the foreign key, sometimes called the base name.

***Note**: Rolenames are also used to model compatibility with legacy data models, where the foreign key was often named differently than the primary key.*

Rolenames migrate across relationships just like any other attributes. For example, suppose that we extend the example to show which PLAYERs have scored in various games throughout the season. The "player-team-id" rolename migrates to the SCORING PLAY entity (along with any other primary key attributes in the parent entity), as shown below.

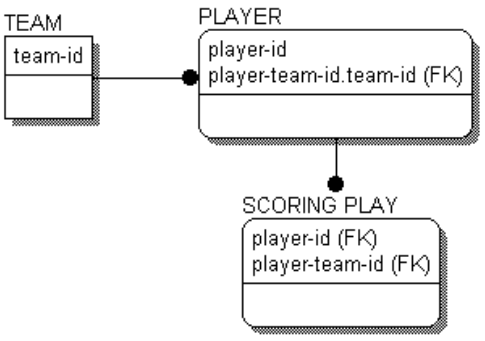



*Diagram Showing Migration of a Rolenamed FK Attribute*

# 4 Naming and Defining Entities and Attributes

## What's In This Chapter?

It is extremely important in data modeling, and in systems development in general, to choose clear and well thought out names for objects. The result of your efforts will be a clear, concise, and unambiguous model of a business area.

Naming standards and conventions are identical for all types of logical models, including both the entity-relationship diagrams and key-based diagrams discussed in previous chapters.

### Chapter Contents

## Naming Entities and Attributes

The most important rule to remember when naming entities is that entity names are always singular. This facilitates reading the model with declarative statements such as “A FLIGHT <transports> zero or more PASSENGERs” and “A PASSENGER <is transported by> one FLIGHT.” When we name an entity, we are also naming each instance. For example, each instance of the PASSENGER entity is an individual passenger, not a set of “passengers.”

Attribute names are singular, too. For example, “person-name,” “employee-SSN,” “employee-bonus-amount” are correctly named attributes. Naming attributes in the singular helps to avoid normalization errors, such as representing more than one fact with a single attribute. The attributes “employee-child-names” or “start-or-end-dates” are plural, and highlight errors in the attribute design.

A good rule of thumb when naming attributes is to use the entity name as a prefix. The rule here is:

- Prefix qualifies.
- Suffix clarifies.

Using this rule, you can easily validate the design and eliminate many common design problems. For example, in the CUSTOMER entity, you can name the attributes “customer-name,” “customer-number,” “customer-address,” etc. If you are tempted to name an attribute “customer-invoice-number,” you use the rule to check that the suffix “invoice-number” tells you more about the prefix “customer.” Since it does not, you must move the attribute to a more appropriate location (such as INVOICE).

You may sometimes find that it is difficult to give an entity or attribute a name without first giving it a definition. As a general principle, providing a good definition for an entity or attribute is as important as providing a good name. The ability to find meaningful names comes with experience and a fundamental understanding of what the model represents.

Because the data model is a description of a business, it is best to choose meaningful business names wherever that is possible. If there is no business name for an entity, you must give the entity a name that fits its purpose in the model.

## Synonyms, Homonyms and Aliases

Not everyone speaks the same language. And we are not always precise in our use of names. Because entities and attributes are identified by their names in a data model, you need to ensure that synonyms are resolved, to ensure that they do not represent redundant data, and then precisely defined, so that each person who reads the model understands which facts are captured in which entity.

It is also important to choose a name that clearly communicates a sense of what the entity or attribute represents. For example, we get a clear sense that there is some difference among things called PERSON, CUSTOMER, and EMPLOYEE. Although they can all represent an individual, they have distinct characteristics or qualities. However, it is the role of the business user to tell you whether or not PERSON and EMPLOYEE are two different things, or just synonyms for the same thing.

Choose your names carefully, and be wary of calling two different things the same if they are not. For example, if you are dealing with a business area which insists on calling its customers "consumers," don't force or insist on the name. You may have discovered an alias, another name for the same thing, or you may have a new "thing" that is distinct from, although similar to, another "thing." In this case, perhaps CONSUMER is a category of CUSTOMER that can participate in relationships that are not available for other categories of CUSTOMER.

ERwin lets you enforce unique naming in the modeling environment. This way you can avoid the accidental use of homonyms (words that are written the same but have different meanings), ambiguous names, or duplication of entities or attributes in the model.

***Note**: Some databases support multiple names in the physical model through the use of defined synonym or alias names, ERwin also supports the definition of synonyms and aliases, but in the physical model only.*

## Entity Definitions

Defining the entities in your logical model is a good way to elaborate on the purpose of the entity, and clarify which facts you want to include in the entity. It is also essential to the clarity of the model. Undefined entities or attributes can be misinterpreted in later modeling efforts, and possibly deleted or unified based on the misinterpretation.

Writing a good definition is more difficult than it might initially seem. Everyone knows what a CUSTOMER is, right? Just try writing a definition of a CUSTOMER that holds up to scrutiny. The best definitions are created using the points of view of many different business users and functional groups within the organization. Definitions that can pass the scrutiny of many, disparate users provide a number of benefits including:

- Clarity across the enterprise.
- Concensus about a single fact having a single purpose.
- Easier identification of "categories," groups of entities that are unique, but have similar purposes or manage similar data.

Most organizations and individuals develop their own conventions or standards for definitions. In practice you will find that long definitions tend to take on a structure that helps the reader to understand the "thing" being defined. Some of these definitions can go on for several pages (CUSTOMER, for example). You may want to adopt the following items as "standards" for the structure of a definition as a starting point, even though IDEF1X and IE do not provide standards for definitions:

- Description
- Business example
- Comments

Each of these components is discussed more fully below.

### Descriptions

A ***description*** should be a clear and concise statement that tells whether an object is or is not the thing you are trying to define. Often such descriptions can be fairly short. Be careful, however, that the description is not too general, or uses terms that have not been defined. Here are a couple of examples, one of good quality, and one which is questionable.

- "A COMMODITY is something that has a value that can be determined in an exchange."

This is a good description because, after reading it, you know that something is a COMMODITY if someone is, or would be, willing to trade something for it. If someone is willing to give us three peanuts and a stick of gum for a marble, then we know that a marble is a COMMODITY.

- "A CUSTOMER is someone who buys something from our company."

This is not a good description. You can easily misunderstand the word "someone" if you know that the company also sells product to other businesses. Also, the business may want to track potential CUSTOMERs, not just those who have already bought something from the company. You could also define "something" more fully to describe whether the sale is of products, services, or some combination of the two.

### Business Examples

It is a good idea to provide typical business examples of the thing being defined, because good examples can go a long way to help the reader understand a definition. Although they are a bit "unprofessional," comments about peanuts and marbles can help a reader to understand the concept of a COMMODITY. The definition said that it had "value." The example can help to show that value is not always "money."

### Comments

You can also include general comments about who is responsible for the definition and who is the source, what state it is in, and when it was last changed as a part of the definition. For some entities, you may also need to explain how it and a related entity or entity name differ. For instance, a CUSTOMER might be distinguished from a PROSPECT.

## Definition References and Circularity

If you open up a dictionary, you may find a situation like this:

- TERM-1 Definition includes reference to, or is based on TERM-2.
- TERM-2 Definition includes reference to, or is based on TERM-3.
- TERM-3 Definition includes reference to, or is based on TERM-1.

The individual definitions look good, but when viewed together are found to be "circular." Without some care, this can happen with entity and attribute definitions. For example:

- CUSTOMER: Someone who buys one or more of our PRODUCTs.
- PRODUCT: Something we offer for sale to CUSTOMERs.

In is important when you define entities and attributes in your data model that you avoid these circular references.

## Constructing a Business Glossary

It is often convenient to make use of common business terms when defining an entity or attribute. For example:

> "A CURRENCY-SWAP is a complex agreement between two PARTYs in which they agree to exchange ***cash flows*** in two different CURRENCYs over a period of time. Exchanges can be fixed over the ***term*** of the swap, or may ***float***. Swaps are often used to ***hedge*** currency and interest ***rate risks***."

In this example, defined terms within a definition are highlighted. Using a style like this makes it unnecessary to define terms each time they are used, since people can look them up whenever needed.

If it will be convenient to use terms that are not the names of entities or attributes, (e.g., common business terms), it is a good idea to provide base definitions of them, and refer to these definitions as would be done for references to entity or attribute definitions. A glossary of commonly used terms, separate from the model, can be used. Such common business terms are highlighted with bold-italics, as in the above passage.

It may seem that a strategy like this will lead initially to a lot of flipping back and forth among definitions. The alternative, however, is to completely define each term every time it is used. If these "internal definitions" appear in many places, they need to be maintained in many places, and the probability that a change will be applied to all of them at the same time is very small.

Developing a glossary of common business terms can serve several purposes. It can become the "base" for use in modeling definitions, and it can, all by itself, be of significant value to the business in helping people to communicate.

## Attribute Definitions

As with entities, it is important to define all attributes clearly. The same rules apply — by comparing a thing to a definition, we should be able to tell if it fits. However, you should beware of things like "account-open-date" defined as, "The date on which the ACCOUNT was opened." You may need to further define what is meant by "opened" before the definition is clear and complete.

Attribute definitions generally should have the same basic structure as entity definitions, including a description, examples, and comments. The definitions should also contain, whenever possible, rules that specify which facts are accepted as valid values for that attribute.

A ***validation rule*** identifies a set of values that an attribute is allowed to take, it constrains or restricts the domain of values that are acceptable. These values have meanings in both an abstract and a business sense. For example, "person-name," if it is defined as the preferred form of address chosen by the PERSON, is constrained to the set of all character strings. You can define any validation rules or valid values for an attribute as a part of the attribute definition. ERwin also lets you assign these validation rules to an attribute using a ***domain***. Supported domains include text, number, datetime, and blob.

Definitions of attributes, such as codes, identifiers, or amounts, often do not lend themselves to good business examples. So, including a description of the attribute's validation rules or valid values is usually a good idea. When defining a validation rule it is good practice to go beyond listing the "values" that an attribute can take. Suppose we define the attribute "customer-status" as follows:

**Customer-status:** *A code that describes the relationship between the CUSTOMER and our business.* **Valid values:** *A, P, F, N*

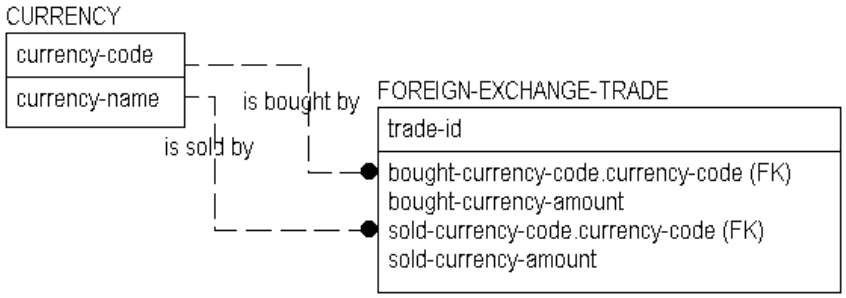
The validation rule specification is not too helpful because it does not define what the codes mean. You can better describe the validation rule using a table or list of values, such as the one below:

| Valid Value | Meaning |
|---|---|
| **A: Active** | The CUSTOMER is currently involved in a purchasing relationship with our company. |
| **P: Prospect** | Someone with which we are interested in cultivating a relationship, but with whom we have no current purchasing relationship. |
| **F: Former** | The CUSTOMER relationship has lapsed — i.e., there has been no sale in the past 24 months. |
| **N: No business accepted** | The company has decided that no business will be done with this CUSTOMER. |

## Rolenames

When a foreign key is contributed to a child entity through a relationship, you may need to write a new or enhanced definition for the foreign key attributes that explains their usage in the child entity. This is certainly the case when the same attribute is contributed to the same entity more than once. These duplicated attributes may appear to be identical, but because they serve two different purposes, they cannot have the same definition.

Consider the example below. Here we see a FOREIGN-EXCHANGE-TRADE with two relationships to CURRENCY.




*Currency Example*

The key of CURRENCY is “currency-code,” (the identifier of a valid CURRENCY that we are interested in tracking). We see from the relationships that one CURRENCY is “bought by,” and one is “sold by” a FOREIGN-EXCHANGE-TRADE.

We see also that the identifier of the CURRENCY (the “currency-code”) is used to identify each of the two CURRENCYs. The identifier of the one that is bought is called “bought-currency-code” and the identifier of the one that is sold is called “sold-currency-code.” These rolenames show that these attributes are not the same thing as “currency-code.”

It would be somewhat silly to trade a CURRENCY for the same CURRENCY at the same time and exchange rate. Thus, for a given transaction (instance of FOREIGN-EXCHANGE-TRADE) “bought-currency-code” and “sold-currency-code” must be different. By giving different definitions to the two rolenames, we can capture the difference between the two currency codes.

| Attribute/Rolename | Attribute Definition |
|---|---|
| **currency-code** | The unique identifier of a CURRENCY. |
| **bought-currency-code** | The identifier (“currency-code”) of the CURRENCY bought by (purchased by) the FOREIGN-EXCHANGE-TRADE. |
| **sold-currency-code** | The identifier (“currency-code”) of the CURRENCY sold by the FOREIGN-EXCHANGE-TRADE. |

The definitions and valididations of the bought and sold codes are based on “currency-code.” “Currency-code” is called a ***base attribute***.

IDEF1X standard dictates that if two attributes with the same name migrate from the same base attribute to an entity, that the attributes must be ***unified***. The result of unification is a single attribute migrated through two relationships. Because of the IDEF1X standard, ERwin automatically unifies foreign key attributes, as well. If you do not want to unify migrated attributes, you can rolename the attributes at the same time that you name the relationship, in ERwin’s Relationship Editor.

## Definitions and Business Rules

Business rules have been mentioned earlier as an integral part of the data model. These rules take the form of relationships, rolenames, candidate keys, defaults, and other modeling structures not yet explored, including generalization categories, referential integrity, and cardinality. And, business rules are also captured in entity and attribute definitions and validation rules.

For example, the CURRENCY entity in the previous figure could be defined either as the set of all valid currencies recognized anywhere in the world, or could be defined as the subset of these which our company has decided to use in its day to day business operations. This is a subtle, but important distinction. In the latter case, there is a business rule, or "policy statement," involved.

This rule manifests itself in the validation rules for "currency-code." It restricts the valid values for "currency-code" to those that are used by the business. Maintenance of the business rule becomes a task of maintaining the table of valid values for CURRENCY. To permit or prohibit trading of CURRENCYs, you simply create or delete instances in the table of valid values.

The attributes "bought-currency-code" and "sold-currency-code" are similarly restricted. And both are further restricted by a validation rule that says "bought-currency-code" and "sold-currency-code" cannot be equal – so each is dependent on the value of the other in its actual use. Using ERwin, validation rules can be addressed in the definitions of attributes, and can also be defined explicitly using validation rules, default values, and valid value lists.

# 5 Refining Model Relationships

## What's In This Chapter?

Relationships are a bit more complex than they might at first seem. They carry a lot of information. Some might say that they are the heart of the data model, because, to a great extent, they describe the rules of the business and the constraints on creating, modifying and deleting instances.

For example, you can use ***cardinality*** to define exactly how many instances are involved in both the child and parent entities in the relationship. And you can further specify how you want to handle database actions such as INSERT, UPDATE, and DELETE using ***referential integrity*** rules.

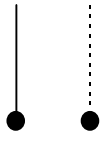
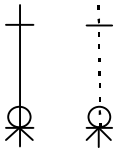
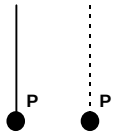
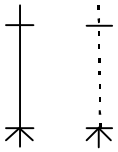
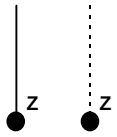
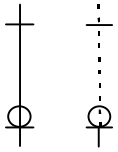
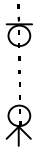
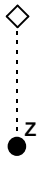
Data modeling also supports highly complex relationship types that enable you construct a logical model of your data that is understandable to both "business" and "systems" experts.

### Chapter Contents

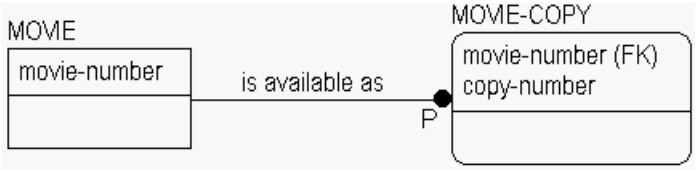
## Relationship Cardinality

Up to this point, we have discussed one-to-many relationships in a logical model, without capturing any information on what we mean by the word "many." The idea of "many" does not mean that there has to be more than one instance of the child connected to a given parent. Instead the "many" in one-to-many really means that there are zero, one or more instances of the child paired up to the parent.

***Cardinality*** is the relational property that defines exactly how many instances appear in a child table for each corresponding instance in the parent table. IDEF1X and IE differ in the symbols are used to specify cardinality. However, both methods provide symbols to denote one or more, zero or more, zero or one, or exactly N, as explained in the following table.

| Cardinality Description | IDEF1X Notation | | IE Notation | |
|---|---|---|---|---|
| | Identifying | Non-identifying | Identifying | Non-identifying |
| One to zero, one, or more |  | |  | |
| One to one or more |  | |  | |
| One to zero or one |  | |  | |
| Zero or one to zero, one, or more (non-identifying only) | |  | |  |
| Zero or one to zero or one (non-identifying only) | |  | |  |

Cardinality lets you specify additional business rules that apply to the relationship. In the example below, the business has decided to identify each MOVIE COPY based on both the foreign key "movie-number" and a surrogate key "copy-number". Further, each MOVIE is available as one or more MOVIE COPYs. The business has also stated that the relationship is identifying, that MOVIE COPY cannot exist unless there is a corresponding MOVIE.




*Cardinality in a One-to-Many Identifying Relationship*

The MOVIE-MOVIE COPY model also specifies the cardinality for the relationship. The relationship line shows that there will be exactly one MOVIE, and only one, participating in a relationship. This is because MOVIE is the parent in the relationship.

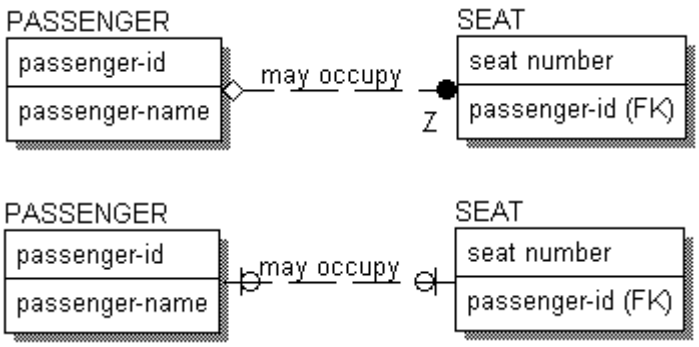
By making MOVIE-COPY the child in the relationship (shown with a dot in IDEF1X), the business defined a MOVIE-COPY as one of perhaps several rentable copies of a movie title. The business also determined that to be included in the database, a MOVIE must have at least one MOVIE-COPY. This makes the cardinality of the "is available as" relationship one-to-one or more. The "P" symbol next to the dot represents cardinality of "one or more." As a result, we also know that a MOVIE with no copies is not a legitimate instance in this database.

In contrast, the business might want to know about all of the MOVIEs in the world, even those for which they have no copies. So their business rule is that for a MOVIE to exist (be recorded in their information system) there can be zero, one, or more copies. To record this business rule, the "P" is removed. When cardinality is not explicitly indicated in the diagram, cardinality is one-to-zero, one or more.

## Cardinality in Non-Identifying Relationships

Non-identifying relationships contribute keys from a parent to a child entity. But, by definition, some (or all) of the keys do not become part of the key of the child. This means that the child will not be identification-dependent on the parent. And there can be situations where an entity at the "many" end of the relationship can exist without a "parent," i.e., it is not existence-dependent.

If the relationship is *mandatory* from the perspective of the child, then the child is existence-dependent on the parent. If it is *optional*, the child is neither existence nor identification-dependent with respect to that relationship (although it may be dependent in other relationships). IDEF1X uses a *diamond* to indicate the optional case, while IE includes a *circle* at the parent end of the relationship line.




*Cardinality in a One-to-Many, Non-Identifying Relationship Using IDEF1X (top) or IE (bottom)*

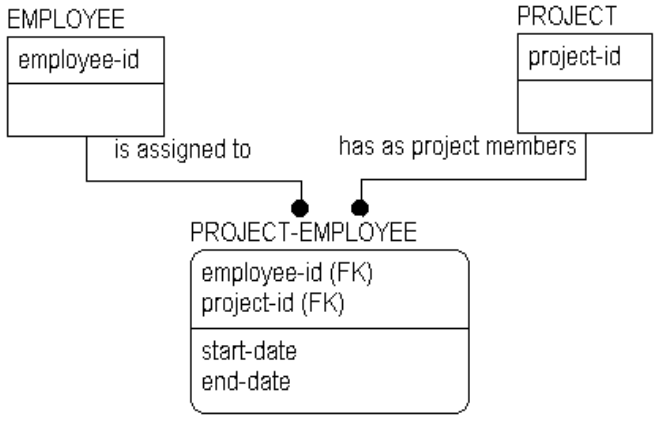
In the example, the attribute “passenger-id” is a foreign key attribute of SEAT. Because the “passenger-id” does not identify the SEAT, it identifies the PASSENGER occupying the SEAT, the business has determined that the relationship is non-identifying. The business has also stated that the SEAT can exist without any PASSENGER, so the relationship is optional. When a relationship is optional, the diagram includes either a diamond in IDEF1X or the circle in IE notation. Otherwise, the cardinality graphics for non-identifying relationships are the same as those for identifying relationships.

The cardinality for the relationship, indicated here with a “Z” in IDEF1X and a single line in IE, states that a PASSENGER <may occupy> zero or one of these SEATs on a flight. Each SEAT can be occupied, in which case the PASSENGER occupying the seat is identified by the “passenger-id”, or it can be unoccupied, in which case the “passenger-id” attribute will be empty (NULL).

## Referential Integrity

Because a relational database relies on data values to implement relationships, the integrity of the data in the key fields is extremely important. If you change a value in a primary key column of a parent table, for example, you must account for this change in each child table in which the column appears as a foreign key. The action that is applied to the foreign key value varies depending on the rules defined by the business.

For example, a business that manages multiple projects might track its employees and projects in a model similar to the one below. The business has determined already that the relationship between PROJECT and PROJECT-EMPLOYEE is identifying, so the primary key of PROJECT becomes a part of the primary key of PROJECT-EMPLOYEE.




*PROJECT-EMPLOYEE Model*

In addition, the business decides that for each instance of PROJECT-EMPLOYEE there is exactly one instance of PROJECT. This means that PROJECT-EMPLOYEE is existence-dependent on PROJECT.

What would happen if you were to delete an instance of PROJECT?

If the business decided that it did not want to track instances in PROJECT-EMPLOYEE if PROJECT is deleted, then you would also have to delete all instances of PROJECT-EMPLOYEE that inherited part of their key from the deleted PROJECT.

The rule that specifies the action taken when a parent key is deleted is called ***referential integrity***. And the referential integrity option chosen for this action in this relationship is ***cascade***. Each time an instance of PROJECT is deleted, this delete cascades to the PROJECT-EMPLOYEE table and causes all related instances in PROJECT EMPLOYEE to be deleted, as well.

Available actions for referential integrity include not only cascade, but also restrict, set null, and set default. Each of the options is explained below:

- **Cascade**. Each time an instance in the parent entity is deleted, each related instance in the child entity must also be deleted.
- **Restrict**. Deletion of an instance in the parent entity is prohibited if there are one or more related instances in the child entity, or deletion of an instance in the child entity is prohibited if there is a related instance in the parent entity.
- **Set Null**. Each time an instance in the parent entity is deleted, the foreign key attribute(s) in each related instance in the child entity are set to NULL.
- **Set Default**. Each time an instance in the parent entity is deleted, the foreign key attribute(s) in each related instance in the child entity are set to the specified default value.
- **<None>**. No referential integrity action is required. Not every action must have a referential integrity rule associated with it. For example, a business may decide that referential integrity is not required when deleting an instance in a child entity. This is a valid business rule in cases where the cardinality is zero or one to zero, one or more, because instances in the child entity can exist even if there are no related instances in the parent entity.

Although referential integrity is not a formal part of the IDEF1X or IE languages, it does capture business rules that indicate how the completed database should work, so it is a critical part of data modeling. Because of this, ERwin provides a method for both capture and display of referential integrity rules.

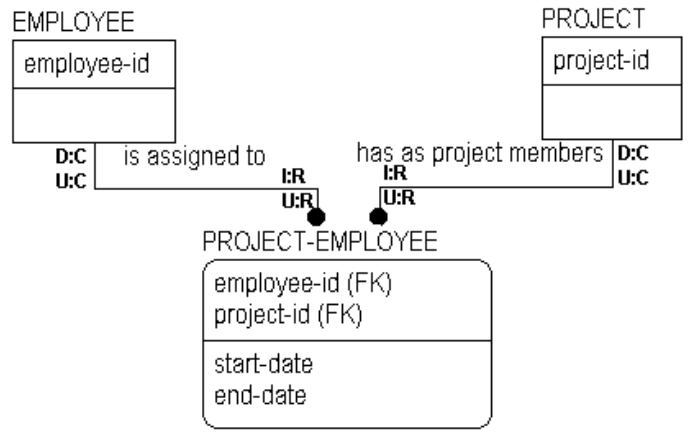
Once referential integrity is defined, the facilitator or analyst should test the referential integrity rules defined by the business users by asking questions or working through different scenarios that show the results of the business decision. When the requirements are defined and fully understood, the facilitator or analyst can recommend specific referential integrity actions, like restrict or cascade.

## Reading Referential Integrity Options

Referential integrity rules vary depending on whether or not the entity is a parent or child in the relationship and the database action that is implemented. As a result, in each relationship there are six possible actions for which referential integrity can be defined. These are:

- PARENT INSERT
- PARENT UPDATE
- PARENT DELETE
- CHILD INSERT
- CHILD UPDATE
- CHILD DELETE

The example below shows referential integrity rules in the EMPLOYEE-PROJECT model.




*Referential Integrity Example*

The referential integrity rules captured in the diagram show the business decision to cascade all deletions in the PROJECT entity to the PROJECT-EMPLOYEE entity. This rule is called PARENT DELETE CASCADE, and is noted in the diagram by the letters "D:C" placed at the parent end of the specified relationship. The first letter in the referential integrity symbol always refers to the database action: I(nsert), U(pdate), or D(elete). The second letter refers to the referential integrity option: C(ascade), R(estrict), SN(set null), and SD(set default).

In the example above, no referential integrity option has been specified for PARENT INSERT, so referential integrity for insert (I:) is not displayed on the diagram.

### RI, Cardinality, and Identifying Relationships

In the previous example, the relationship between PROJECT and PROJECT-EMPLOYEE is identifying. So the valid options for referential integrity for the parent entity in the relationship, PROJECT, include cascade and restrict.

Cascade indicates that all instances of PROJECT-EMPLOYEE that are affected by the deletion of an instance of PROJECT should also be deleted. Restrict indicates that a PROJECT cannot be deleted until all instances of PROJECT-EMPLOYEE that have inherited its key have been deleted. If there are any left, the delete is “restricted.”

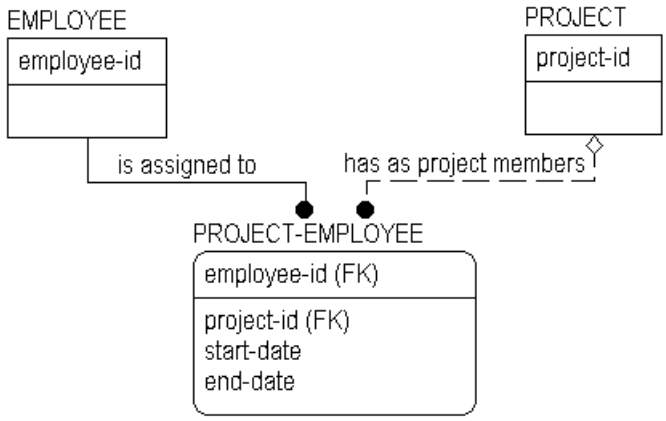
Why would we want to restrict the deletion? One reason might be that the business needs to know other facts about a PROJECT-EMPLOYEE such as the “date started” on the project. If you cascade the delete, you lose this supplementary information.

In the case of updating an instance in the parent entity, the business has also determined that the updated information should cascade to the related instances in the child entity.

As you can see in the example, different rules apply when an instance is inserted, updated, or deleted in the child entity. When an instance is inserted, for example, the action is set to restrict. This rule appears in as "I:R" placed next to the child entity in the relationship. This means that an instance can be added to the child entity *only* if the referenced foreign key matches an existing instance in the parent entity. So, you can insert a new instance in PROJECT-EMPLOYEE only if the value in the key field matches a key value in the PROJECT entity.

## RI, Cardinality, and Non-Identifying Relationships

If the business decides that PROJECT-EMPLOYEEs are not existence or identification-dependent on PROJECT, you can change the relationship between PROJECT and PROJECT-EMPLOYEE to optional, non-identifying. In this type of relationship, the referential integrity options are very different.




*Referential Integrity for a Non-Identifying Relationship*

Because a foreign key contributed across a non-identifying relationship is allowed to be NULL, one of the referential integrity options you could specify for PARENT DELETE is "set null." ***Set null*** indicates that if an instance of PROJECT is deleted, then any foreign key inherited from PROJECT in a related instance in PROJECT-EMPLOYEE should be set to NULL. The delete does not cascade as in our previous example, and it is not prohibited (as in restrict). The advantage of this approach is that you can preserve the information about the PROJECT-EMPLOYEE while effectively breaking the connection between the PROJECT-EMPLOYEE and PROJECT.

Decisions to use cascade or set null reflect business decisions about maintaining the “historical” knowledge of relationships represented by the foreign keys.

## Additional Relationship Types

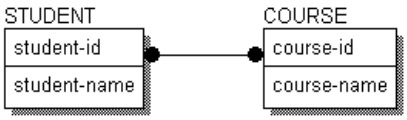
Relationships define whether the child entity is dependent or independent of the parent entity and how many instances are related in parent and child entities. As you develop a logical model, you may find relationships that do not fall into the standard, one-to-many relationships discussed in previous chapters. These relationships include:

- **Many-to-many relationships.** A relationship where one entity <owns> many instances of a second entity, and the second entity also <owns> many instances of the first entity. For example, an EMPLOYEE <has> one or more JOB TITLEs, and a JOB TITLE <is applied to> one or more EMPLOYEEs.
- **N-ary relationships**. A simple one-to-many relationship between two entities is termed binary. When a one-to-many relationship exists between two or more parents and a single child entity it is termed an ***n-ary relationship***.
- **Recursive relationships.** Entities that have a relationship to themselves take part in recursive relationships. For example, for the EMPLOYEE entity, you could include a relationship to show that One EMPLOYEE <manages> one or more EMPLOYEEs. This type of relationship is also used for bill-of-materials structures, to show relationships between parts.
- **Subtype relationships**. Related entities are grouped together so that all common attributes appear in a single entity, but all attributes that are not in-common appear in separate, related entities. For example, the EMPLOYEE entity could be subtyped into FULL-TIME and PART-TIME.

Each of these relationship types is discussed more fully later in this chapter.

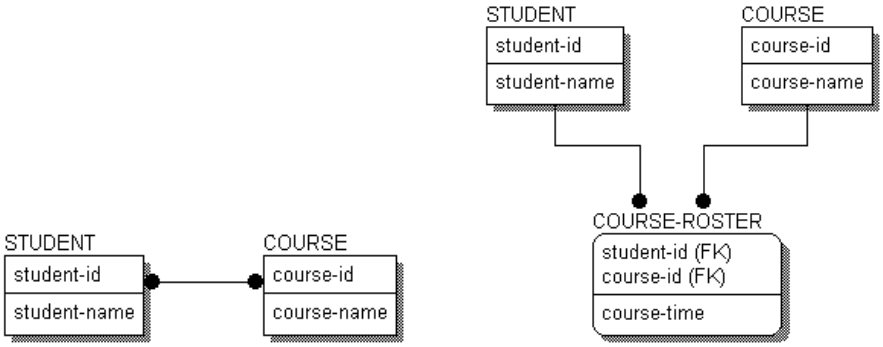
## Many-to-Many Relationships

In key-based and fully-attributed models, relationships must relate zero or one instances in a parent entity to a specific set of instances in a child entity. As a result of this rule, many-to-many relationships that were dicovered and documented in an ERD or earlier modeling phase must be broken down into a pair of one-to-many relationships.




The example above shows a many-to-many relationship between STUDENTs and COURSEs. If you did not eliminate the many-to-many relationship between COURSE and STUDENT, for example, the key of COURSE would be included in the key of STUDENT, and vice versa. But COURSEs are identified by their own keys, and likewise for STUDENTs -- creating an endless loop!

You can eliminate the many-to-many relationship by creating an ***associative entity***. In the example below, we have resolved the many-to-many relationship between STUDENT and COURSE by adding COURSE-ROSTER entity.



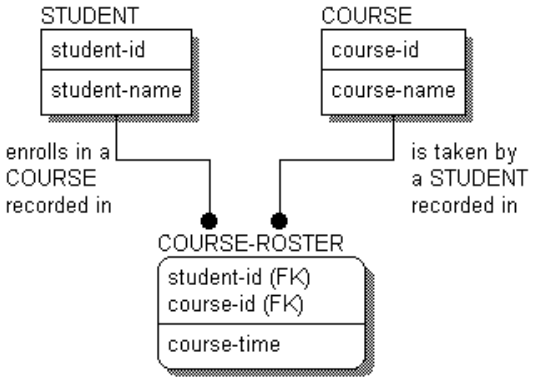

*Resolving the STUDENT-COURSE Relationship Using an Associative Entity*

COURSE-ROSTER is an ***associative entity***, meaning it is used to define the association between two related entities.

Many-to-many relationships often hide meaning. In the diagram with a many-to-many relationship, you know that a STUDENT enrolls in many COURSEs, but no information is included to show how. When you resolve the many-to-many relationship, you see not only how the entities are related, but uncover additional information, such as the "course-time," that also describes facts about the relationship.

Once the many-to-many relationship is resolved, you are faced with the requirement to include relationship verb phrases that validate the structure. There are two ways to do this: construct new verb phrases, or use the verb phrases as they existed for the many-to-many relationship. The most straightforward way is to continue to read the "many-to-many" relationship, through the associative entity. So you can read A STUDENT <enrolls in> many COURSEs and A COURSE <is taken by> many STUDENTs. Many modelers adopt this style for constructing and reading a model.

There is another style, which is equally correct, but a bit more cumbersome. The structure of the model is exactly the same, but the verb phrases are different, and the model is "read" in a slightly different way. In this example, you would read: A STUDENT <enrolls in a COURSE recorded in> one or more COURSE-ROSTERs, and A COURSE <is taken by a STUDENT recorded in> one or more COURSE-ROSTERs.



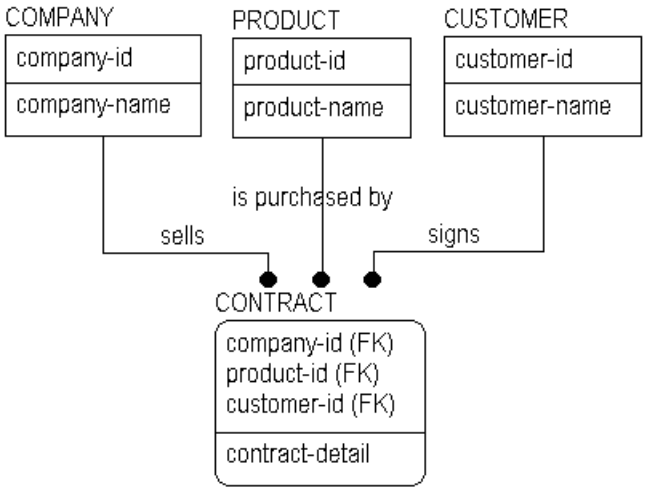

*Reading Relationships Through Associative Entities*

Although the verb phrases have gotten fairly long, the reading follows the standard pattern reading directly from the parent entity to the child.

Whichever style you choose, be consistent. Deciding how to record verb phrases for many-to-many relationships is not too difficult when the structures are fairly simple, as in our examples. However, this can become more difficult when the structures become more complex, such as when the entities on either side of the associative entities are themselves associative entities, which are there to represent other many-to-many relationships.

## N-ary Relationships

When a single parent-child relationship exists, the relationship is called ***binary***. All of the previous examples of relationships to this point have been binary relationships. However, when creating a data model, it is not uncommon to come across ***n-ary relationships***, which are the modeling name for relationships between two or more parent entities and a single child table. An example of an n-ary relationship is shown below.

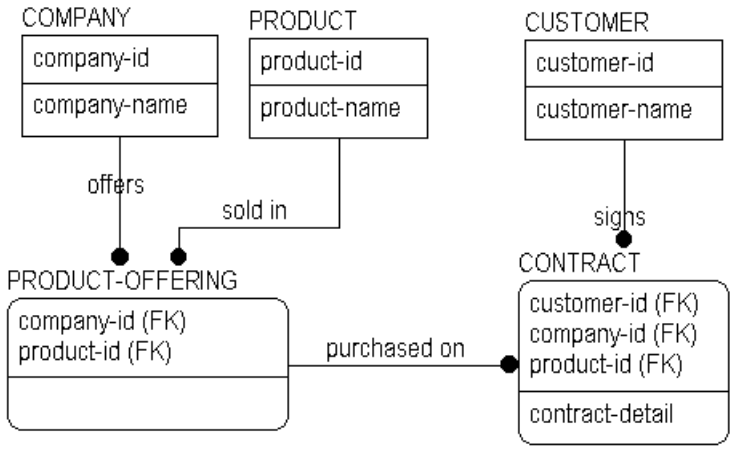



*N-ary Relationship*

Like many-to-many relationships, three-, four-, or "n-ary" relationships are valid constructs in entity-relationship diagrams. Also like many-to-many relationships, n-ary relationships should be resolved in later models using a set of binary relationships to an associative entity.

If you consider the business rule stated in the example above, you see that a CONTRACT represents a three-way relationship among COMPANY, PRODUCT, and CUSTOMER. The structure indicates that many COMPANYs sell many PRODUCTs to many CUSTOMERs. When you see a relationship like this, however, you know that there are business questions begging to be asked. For example, "Must a product be offered by a company before it can be sold?" "Can a customer establish a single contract including products from several different companies?" and, "Do we need to keep track of which customers "belong to" which companies?" Depending on the answers, the structures may change.

If, for example, the answer to the question "Must a product be offered by a company before it can be sold?" is “yes,” then we would have to change the structure as shown below.




*Resolving an N-ary Relationship*

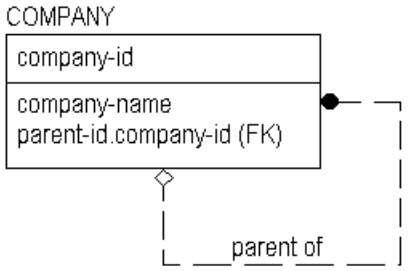
Because PRODUCTs must be offered by COMPANYs, you can create an associative entity to capture this relationship. As a result, the original “three-way” relationship to CONTRACT is replaced by two, “two-way” relationships.

By asking a variety of business questions, it is likely that you will find that most “n-ary” relationships can be broken down into a series of relationships to associative entities.

## Recursive Relationships

An entity can participate in a recursive relationship (also called "fish hook") in which the same entity is both the parent and the child. This relationship is an important one when modeling data originally stored in legacy DBMSs such as IMS or IDMS that use recursive relationships to implement bill of materials structures.

For example, a COMPANY can be the “parent of” other COMPANYs. As with all non-identifying relationships, the key of the parent entity appears in the data area of the child entity.




*Recursive Relationship Example*

The recursive relationship for COMPANY includes the diamond symbol to indicate that the foreign key can be NULL, such as when a COMPANY has no parent. Recursive relationships must be both optional (diamond) and non-identifying.

The “company-id” attribute migrated through the recursive relationship, and appears in the example with the rolename “parent-id”. The reason for this is two-fold. First, as a general design rule, an attribute cannot appear twice in the same entity under the same name. Thus, to complete a recursive relationship, you must provide a rolename for the migrated attribute.

Second, the attribute “company-id” in the key, which identifies each instance of COMPANY, is not the same thing as the “company-id” migrated through the relationship, which identifies the parent COMPANY. You cannot use the same definition for both attributes, so the migrate attribute must be rolenamed. An example of possible definitions appears below:

| | |
|---|---|
| **company-id:** | The unique identifier of a COMPANY. |
| **parent-id:** | The “company-id” of the parent COMPANY. Not all COMPANYs have a parent COMPANY. |

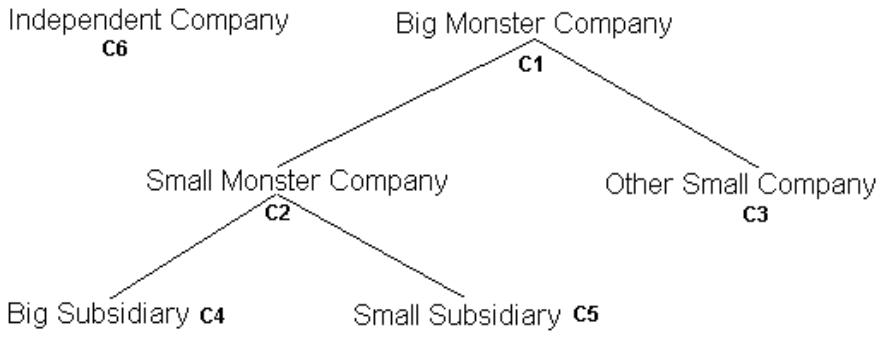
If you create a sample instance table, such as the one below, you can test the rules in the relationship to ensure that they are valid.

**COMPANY**

| company-id | parent-id | company-name |
|---|---|---|
| C1 | NULL | Big Monster Company |
| C2 | C1 | Smaller Monster Company |
| C3 | C1 | Other Smaller Company |
| C4 | C2 | Big Subsidiary |
| C5 | C2 | Small Subsidiary |
| C6 | NULL | Independent Company |

*Sample Instance Table for COMPANY*

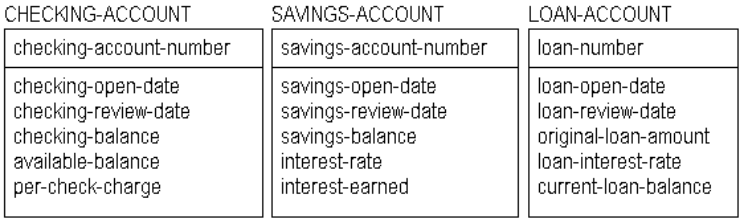
The sample instance table shows that “Big Monster Company” is parent of “Smaller Monster Company” and “Other Smaller Company.” “Smaller Monster Company,” in turn, is parent of “Big Subsidiary” and “Small Subsidiary.” “Independent Company” is not the parent of any other, and has no parent. “Big Monster Company” also has no parent. If you diagram this information hierarchically, you can validate the information in the table.




*COMPANY Hierarchy*

## Subtype Relationships

A ***subtype relationship***, also referred to as a generalization category, generalization hierarchy, or inheritance hierarchy, is a way to group a set of entities that share common characteristics. For example, we might find during a modeling effort, that several different types of ACCOUNTs exist in a bank, such as checking, savings and loan accounts, as shown below.



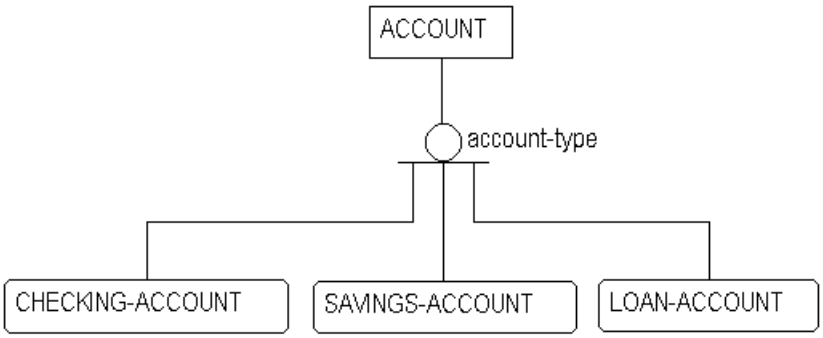

*Example Account Entities*

When you recognize similarities among the different independent entities, you may be able to collect together attributes common to all three types of accounts into a hierarchical structure.

You can move these common attributes into a higher level entity called the ***supertype entity*** (or generalization entity). Those that are specific to the individual account types remain in the ***subtype entities***. In the example, you can create a supertype entity called ACCOUNT to represent the information that is common across the three types of accounts. The supertype ACCOUNT includes a primary key of "account-number."

Three ***subtype*** entities, CHECKING-ACCOUNT, SAVINGS-ACCOUNT, and LOAN-ACCOUNT, are added as dependent entities that are related to ACCOUNT using a subtype relationship.

The result is a structure like the one shown below.




*Subtype Relationship Example*

In this example, an ACCOUNT is either a CHECKING ACCOUNT, a SAVINGS-ACCOUNT, or a LOAN-ACCOUNT. Each subtype entity is an ACCOUNT, and inherits the properties of ACCOUNT. The three different subtype entities of ACCOUNT are mutually exclusive.

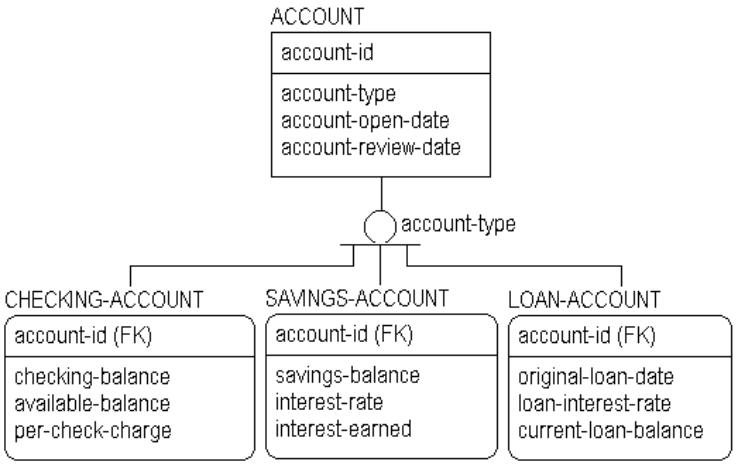
In order to distinguish one type of ACCOUNT from another, we add the attribute "account-type" as the ***subtype discriminator***. The subtype discriminator is an attribute of the category supertype (ACCOUNT) and its value will tell us which type of ACCOUNT we have.

Once you have established the subtype relationship, you can examine each attribute in the original model, in turn, to determine if it should remain in the subtype entities, or move to the supertype. For example, each subtype entity has an "open-date." If the definitions of these three kinds of "open-date" are the same, you can move them to the supertype, and drop them from the subtype entities.

You need to analyze each attribute in turn to determine if it remains in the subtype entity or moves to the supertype. In those cases where a single attribute appears in most, but not all, of the subtype entities, you face a more difficult decision. You can either leave the attribute with the subtype entities, or move the attribute up to the supertype. If this attribute appears in the supertype, this will mean that the value of the attribute in the supertype will be NULL when the attribute is one not included in the corresponding subtype entity.

Which alternative to choose depends on how many of the subtype entities share the common attribute. If most do, it is good practice, at higher level models, to move them up. If few subtype entities share the attribute, it is best to leave them where they are. In lower level models, depending on the purpose, it is often appropriate to leave the attributes in their subtype entity.

After analysis, the resulting model might appear like the one below.



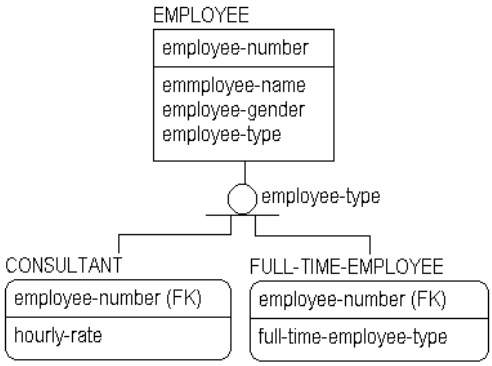

*Account Subtype Example*

When developing a subtype relationship, you must also be aware of any specific business rules that need to be imposed at the subtype level that are not pertinent to other subtypes of the supertype. For example, LOAN accounts are deleted after they are paid-up. You would hardly like to delete CHECKING and SAVINGS accounts under the same conditions.

There may also be relationships that are meaningful to a single subtype, and not to any other subtype in the hierarchy. The LOAN entity needs to be examined, for example, to ensure that any previous relationships to records of customer payments or assets are not lost because of a different organizational structure.

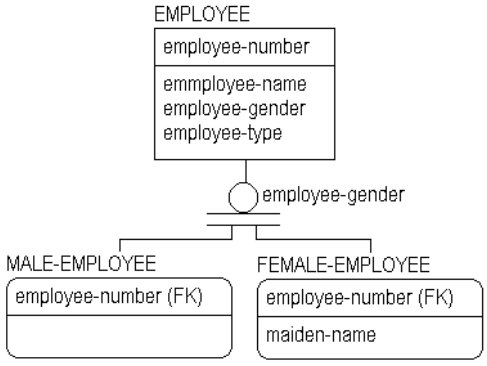
## Complete Versus Incomplete Subtype Structures

In IDEF1X, different symbols are used to specify whether or not the set of subtype entities in a subtype relationship is fully defined. An incomplete subtype indicates that the modeler feels there may be other subtype entities that have not yet been discovered, and is indicated by a *single line* at the bottom of the subtype symbol.




*Incomplete Subtype*

A complete subtype indicates that the modeler is certain that all possible subtype entities are included in the subtype structure. For example, a complete subtype could capture information specific to male and female employees, as shown below. A complete subtype is indicated by *two lines* at the bottom of the subtype symbol.




*Complete Subtype*

When you create a subtype relationship, it is a good rule of thumb to also create a validation rule for the discriminator. This helps to ensure that all subtypes have been discovered. For example, a validation rule for "account-type" might include: C=checking account, S=savings account, L=loans. If the business also has legacy data with account types of "O," the validation rule uncovers the undocumented type and lets you decide if the "O" is a symptom of poor design in the legacy system, or a real account type that you forgot.

### Inclusive and Exclusive Relationships

Unlike IDEF1X, IE notation does not distinguish between complete and incomplete subtype relationships. Instead, IE notation documents whether the relationship is *exclusive* or *inclusive*.

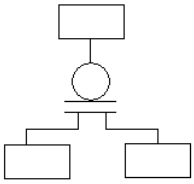
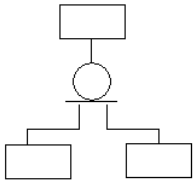
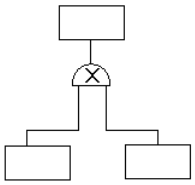
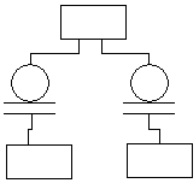
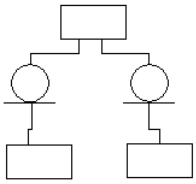
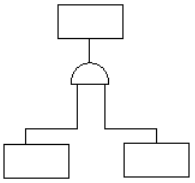
In an exclusive subtype relationship, each instance in the supertype can relate to one and only one subtype. For example, you might model a business rule says that an employee can be either a full-time or part-time employee, but not both. To create the model, you would include an EMPLOYEE supertype entity with FULL-TIME and PART-TIME subtype entities, and a discriminator attribute called "employee-status." In addition, you would constrain the value of the discriminator to show that valid values for it include "F" to denote full-time and "P" to denote part-time.

In an inclusive subtype relationship, each instance in the supertype can relate to one or more subtypes. In our example, the business rule might now state that an employee could be full-time, part-time, or both. In this example, you would constrain the value of the discriminator to show that valid values for it include "F" to denote full-time, "P" to denote part-time, and "B" to denote both.

***Note**: In IDEF1X notation, you can represent inclusive subtypes by drawing a separate relationship between the supertype entity and each subtype entity.*

## IDEF1X and IE Subtype Notation

The following chart illustrates subtype notation in IDEF1X and IE.

| | IDEF1X Subtype Notation | | |
|---|---|---|---|
| | Complete | Incomplete | IE Subtype Notation |
| **Exclusive Subtype** |  |  |  |
| **Inclusive Subtype** |  |  |  |

### When to Create a Subtype Relationship

To summarize, there are three reasons to create a subtype relationship:

- First, the entities share a common set of attributes. This was the case in our examples above.
- Second, the entities share a common set of relationships. We have not explored this, but, referring back to our account structure, we could as needed, collect any common relationships that the subtype entities had into a single relationship from the generic parent. For example, if each account type is related to many CUSTOMERs, you can include a single relationship at the ACCOUNT level, and eliminate the separate relationships from the individual subtype entities.
- Third, subtype entities should be exposed in a model if the business demands it (usually for communication or understanding purposes) even if the subtype entities have no attributes that are different, and even if they participate in no relationships distinct from other subtype entities. Remember that one of the major purposes of a model is to assist in communication of information structures, and if showing subtype entities assists with this, then show them.

# Normalization

## Introduction

Normalization is the process of making a database design comply with the design rules outlined by E. F. Codd for relational databases. Following the rules for normalization, you can control and eliminate data redundancy by removing all model structures that provide multiple ways to know the same fact.

The goal of normalization is to ensure that there is only one way to know a "fact." A useful slogan summarizing this goal is:

**ONE FACT IN ONE PLACE!**

To provide a basic understanding of the principles of normalization, this chapter includes a variety of examples of common design problems and normalization solutions.

### Chapter Contents

## Overview of the Normal Forms

The following are formal definitions for the most common normal forms.

### Functional Dependence (FD)

Given an entity E, attribute B of E is functionally dependent on attribute A of E if and only if each value of A in E has associated with it precisely one value of B in E (at any one time). In other words, A uniquely determines B.

### Full Functional Dependence

Given an entity E, an attribute B of E is fully functionally dependent on a set of attributes A of E if and only if B is functionally dependent on A and not functionally dependent on any proper subset of A.

### First Normal Form (1NF)

An entity E is in 1NF if and only if all underlying values contain atomic values only. Any repeating groups (that might be found in legacy COBOL data structures, for example) must be eliminated.

### Second Normal Form (2NF)

An entity E is in 2NF if it is in 1NF and every non-key attribute is fully dependent on the primary key. In other words, there are no partial key dependencies — dependence is on the entire key K of E, and not on a proper subset of K.

### Third Normal Form (3NF)

An entity E is in 3NF if it is in 2NF and no non-key attribute of E is dependent on another non-key attribute. There are several equivalent ways to express 3NF. Here is a second: An entity E is in 3NF if it is in 2NF and every non-key attribute is non transitively dependent on the primary key. A third and final way is: An entity E is in 3NF if every attribute in E carries a fact about all of E (2NF) and only about E (as represented by the entity's entire key and only by that key). One way to remember how to implement 3NF is using the following quip: "Each attribute relies on the key, the whole key, and nothing but the key, so help me Codd!"

Beyond 3NF lie three more normal forms, Boyce-Codd, Fourth and Fifth. In practice, third normal form is the standard. At the level of the physical database design, choices are usually made to "denormalize" a structure in favor of performance for a certain set of transactions. This may introduce redundancy in the structure, but is often worth it.

## Common Design Problems

Many common design problems are a result of violating one of the normal forms. Common problems include:

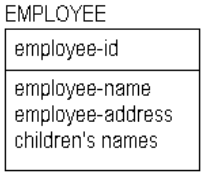
- Repeating data groups
- Multiple use of the same attribute
- Multiple occurrences of the same fact
- Conflicting facts
- Derived attributes
- Missing information

These problems are examined individually in the follow sections, and explained using models and sample instance data. When you work on eliminating design problems, the use of sample instance data can be invaluable in discovering many normalization errors.

### Repeating Data Groups

Repeating data groups can be defined as lists, repeating elements, or internal structures inside an attribute. This structure, although common in legacy data structures, violates first normal form and must be eliminated in an RDBMS model. This is because an RDBMS cannot handle variable-length repeating fields, because it offers no ability to subscript through arrays of this type. The entity below contains a repeating data group, “children’s-names”. Repeating data groups violate first normal form, which basically states that “An entity is in first normal form if each of its attributes has a single meaning and not more than one value for each instance.”

Repeating data groups, such as in the example below, present problems when defining a database to contain the actual data. For example, after designing the EMPLOYEE entity, you are faced with the questions “How many children’s names do we need to record?” “How much space should we leave in each row in the database for the names?” and “What will we do if we have more names than remaining space?”




*EMPLOYEE Entity*

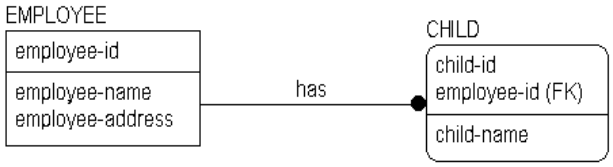
A sample instance table might clarify the problem:

**EMPLOYEE**

| emp-id | emp-name | emp-address | children's-names |
|---|---|---|---|
| E1 | Tom | Berkeley | Jane |
| E2 | Don | Berkeley | Tom, Dick, Donna |
| E3 | Bob | Princeton | — |
| E4 | John | New York | Lisa |
| E5 | Carol | Berkeley | — |

*EMPLOYEE Sample Instance Table*

In order to fix the design, we must somehow remove the list of children's names from the EMPLOYEE entity. One way to do this is to add a CHILD table to contain the information about employee's children. Once that is done, you can represent the names of the children as single entries in the CHILD table. In terms of the physical record structure for employee, this can resolve some of your questions about space allocation, and prevent wasting space in the record structure for employees who have no children or, conversely, deciding how much space to allocate for employees with families.




**EMPLOYEE**

| emp-id | emp-name | emp-address |
|---|---|---|
| E1 | Tom | Berkeley |
| E2 | Don | Berkeley |
| E3 | Bob | Princeton |
| E4 | Carol | Berkeley |

**CHILD**

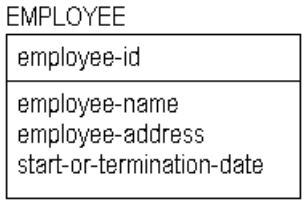
| emp-id | child-id | child-name |
|---|---|---|
| E2 | C1 | Tom |
| E2 | C2 | Dick |
| E2 | C3 | Donna |
| E4 | C1 | Lisa |

*Sample Instance Tables for the EMPLOYEE-CHILD Model*

This change makes the first step toward a normalized model – conversion to first normal form. Both entities now contain only fixed-length fields, which are easy to understand and program.

## Multiple Use of the Same Attribute

It is also a problem when a single attribute can represent one of two facts, and there is no way to understand which fact it represents. For example, the EMPLOYEE entity below contains the attribute "start-or-termination-date" in which you can record this information for an employee.




*EMPLOYEE Entity with "Start-or-termination-date" Attribute*

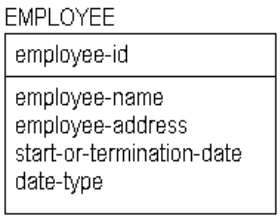
**EMPLOYEE**

| emp-id | emp-name | emp-address | start-or-termination-date |
|---|---|---|---|
| E1 | Tom | Berkeley | Jan 10. 1998 |
| E2 | Don | Berkeley | May 22, 1998 |
| E3 | Bob | Princeton | Mar 15, 1997 |
| E4 | John | New York | Sep 30, 1998 |
| E5 | Carol | Berkeley | Apr 22, 1994 |
| E6 | George | Pittsburgh | Oct 15, 1998 |

*Sample Instance Table Showing "Start-or-termination-date"*

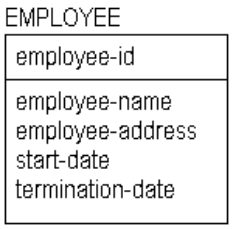
The problem in the current design is that there is no way to record both a start date, "the date that the EMPLOYEE started work," and a termination date, "the date on which an EMPLOYEE left the company," in situations where both dates are known. This is because a single attribute represents two different facts. This is also a common structure in legacy COBOL systems, but one that often resulted in maintenance nightmares and misinterpretation of information.

The solution is to allow separate attributes to carry separate facts. Below is an attempt to correct the problem. It's still not quite right. To know the start date for an employee, for example, you have to derive what kind of date it is from the "date-type" attribute. While this may be efficient in terms of physical database space conservation, it wreaks havoc with query logic.




In fact, this "solution" actually creates a different type of normalization error, because "date-type" does not depend on "employee-id" for its existence. This is also poor design because it solves a technical problem, but does not solve the underlying business problem – how to store two facts about an employee.

When you analyze the data, you can quickly determine that a better solution is to let each attribute carry a separate fact.




*EMPLOYEE Entity with "Start-date" and "Termination-date" Attributes*

**EMPLOYEE**

| emp-id | emp-name | emp-address | start-date | termination-date |
|---|---|---|---|---|
| E1 | Tom | Berkeley | Jan 10. 1998 | — |
| E2 | Don | Berkeley | May 22, 1998 | — |
| E3 | Bob | Princeton | Mar 15, 1997 | — |
| E4 | John | New York | Sep 30, 1998 | — |
| E5 | Carol | Berkeley | Apr 22, 1994 | — |
| E6 | George | Pittsburgh | Oct 15, 1998 | Nov 30, 1998 |

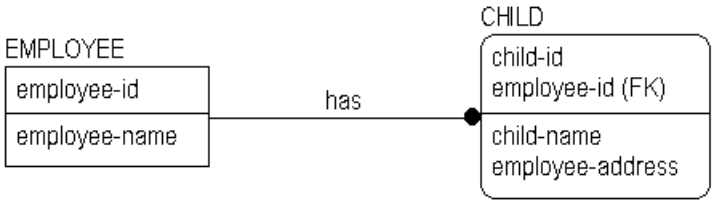
*Sample Instance Table Showing "Start-date" and "Termination-date"*

Each of the two previous situations contained a first normal form error. By changing the structures we have made sure that an attribute appears only once in the entity, and that it carries only a single fact. If you make sure that all entity and attribute names are singular, and that no attribute can carry multiple facts, then you will have taken a large step toward assuring that a model is in first normal form.

## Multiple Occurrences of the Same Fact

One of the goals of a relational database is to maximize data integrity, to ensure that the information contained in the database is correct and that facts within the database do not conflict. To maximize data integrity, it is important to represent each fact in the database once and only once. If a fact appears in two or more places, errors can begin creep into the data. The only exception to this rule (one fact in one place) is in the case of key attributes, which appear multiple times in a database. The integrity of keys, however, is managed using referential integrity, which is discussed earlier in this book.

Multiple occurrences of the same fact often point to a flaw in the original database design. In the example below, you can see that including "employee-address" in the CHILD entity has introduced an error in the database design. If an employee has multiple children, the address must be maintained separately for each child.

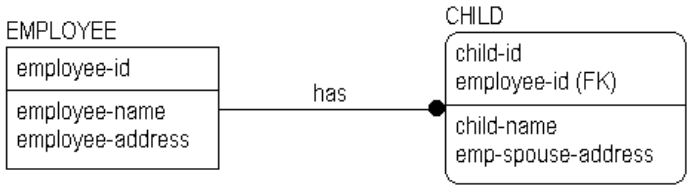



*Multiple Occurrences of the Same Fact*

"Employee-address" is information about the EMPLOYEE, not information about the CHILD. In fact, this model violates *second normal form*, which states that each fact must depend on the entire key of the entity in order to belong to the entity. The example above is not in second normal form because "employee-address" does not depend on the entire key of CHILD, only on the "employee-id" portion, creating a partial key dependency. If you place "employee-address" back with EMPLOYEE, you can ensure that the model is in at least second normal form.

## Conflicting Facts

Conflicting facts can occur for a variety of reasons, including violation of first, second or third normal forms. An example of conflicting facts occurring through a violation of second normal form appears below:




*EMPLOYEE-CHILD Model With Conflicting Facts*

### EMPLOYEE

| emp-id | emp-name | emp-address |
|---|---|---|
| E1 | Tom | Berkeley |
| E2 | Don | Berkeley |
| E3 | Bob | Princeton |
| E4 | Carol | Berkeley |

### CHILD

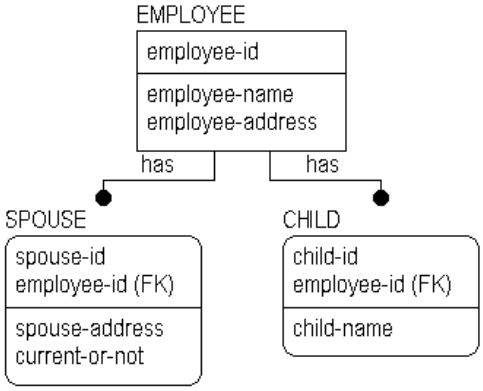
| emp-id | child-id | child-name | emp-spouse-address |
|---|---|---|---|
| E1 | C1 | Jane | Berkeley |
| E2 | C1 | Tom | Berkeley |
| E2 | C2 | Dick | Berkeley |
| E2 | C3 | Donna | Cleveland |
| E4 | C1 | Lisa | New York |

*Sample Instance Tables Showing "Emp-spouse-address"*

The attribute named "emp-spouse-address" is included in CHILD, but this design is a second normal form error. The instance data highlights the error. As you can see, Don is the parent of Tom, Dick, and Donna but the instance data shows two different addresses recorded for Don's spouse. Perhaps Don has two spouses (one in Berkeley, and one in Cleveland), or Donna has a different mother from Tom and Dick. Or perhaps Don has one spouse with addresses in both Berkeley and Cleveland. Which of these is the answer? There is no way to know from the model as it stands. Business users are the only source that can eliminate this type of semantic problem, so analysts need to ask the right questions about the business to uncover the correct design.

The problem in the example is that “emp-spouse-address” is a fact about the EMPLOYEE’s SPOUSE, not about the CHILD. If we leave the structure the way it currently is, then every time Don’s spouse changes address (along with Don, we presume), we will have to update that fact in multiple places; once in each CHILD instance where Don is the parent. And if we have to do that in multiple places, what is the chance that we will always get it right everywhere? Not too good.

Once it is recognized that “emp-spouse-address” is a fact not about a child but about a spouse, you can correct the problem. To capture this information, you can add a SPOUSE entity to the model.




*Spouse Entity Added to the EMPLOYEE-CHILD Model*

**EMPLOYEE**

| emp-id | emp-name | emp-address |
|---|---|---|
| E1 | Tom | Berkeley |
| E2 | Don | Berkeley |
| E3 | Bob | Princeton |
| E4 | Carol | Berkeley |

*Sample Instance Tables*

**CHILD**

| emp-id | child-id | child-name |
|---|---|---|
| E1 | C1 | Jane |
| E2 | C1 | Tom |
| E2 | C2 | Dick |
| E2 | C3 | Donna |
| E4 | C1 | Lisa |

**SPOUSE**

| emp-id | spouse-id | spouse-address | current-spouse |
|---|---|---|---|
| E2 | S1 | Berkeley | Y |
| E2 | S2 | Cleveland | N |
| E3 | S1 | Princeton | Y |
| E4 | S1 | New York | Y |
| E5 | S1 | Berkeley | Y |

*Sample Instance Tables Showing the SPOUSE Entity*

In breaking out SPOUSE into a separate entity, you can see that the data for Don's spouse's address is correct -- Don just had two spouses, one current and one former.

By making sure that every attribute in an entity carries a fact about *that* entity, you can generally be sure that a model is in at least second normal form. Further transforming a model into third normal form generally reduces the likelihood that the database will become corrupt, i.e., that it will contain conflicting information, or that required information will be missing.

## Derived Attributes

Another example of conflicting facts occurs when third normal form is violated. For example, if you included both a “birth-date” and an “age” attribute as non-key attributes in the CHILD entity, you violate third normal form. This is because “age” is *functionally dependent* on “birth-date.” By knowing “birth-date” and the date today, we can *derive* the “age” of the CHILD.

***Derived attributes*** are those that may be computed from other attributes (e.g., totals) and therefore need not be stored directly. To be accurate, derived attributes need to be updated every time their derivation source(s) is updated. This creates a large overhead in an application that does batch loads or updates, for example, and puts the responsibility on application designers and coders to ensure that the updates to derived facts are performed.

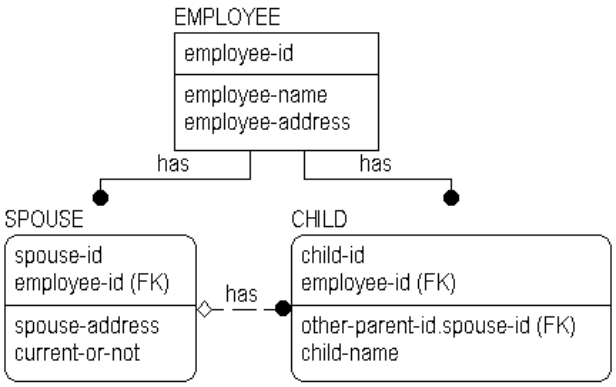
A goal of normalization is to ensure that there is only one way to know each fact recorded in the database. If we know the value of a derived attribute, and we know the algorithm by which it is derived and the values of the attributes used by the algorithm, then there are two ways to know the fact (look at the value of the derived attribute, or derive it from scratch). If you can get an answer two different ways, it is possible that the two answers will be different.

For example, we can choose to record both the “birth-date” and the “age” for CHILD. And suppose that the “age” attribute is only changed in the database during an end of month maintenance job. Then, when we ask the question, “How old is such and such CHILD?” we can directly access “age” and get an answer, or we can, at that point, subtract “birth-date” from “today’s-date.” If we did the subtraction, we would *always* get the right answer. If “age” has not been updated recently, it might give us the wrong answer, and there would *always* be the *potential for conflicting answers*.

There are situations, where it makes sense to record derived data in the model, particularly if the data is expensive to compute. It can also be very useful in discussing the model with the business. Although the theory of modeling says that you should never include derived data (and we urge you to do so only sparingly), break the rules when you must. But at least record the fact that the attribute is derived and state the derivation algorithm.

## Missing Information

Missing information in a model can sometimes result from efforts to normalize the data. In our example, adding the SPOUSE entity to the EMPLOYEE-CHILD model improves the design, but destroys the implicit relationship between the CHILD entity and the SPOUSE address. It is possible that the reason that "emp-spouse-address" was stored in the CHILD entity in the first place was to represent the address of the other parent of the child (which was assumed to be the spouse). If we need to know the other "parent" of each of the children, then we must add this information to the CHILD entity.




*Replacing Missing Information Using a New Relationship*

### EMPLOYEE

| emp-id | emp-name | emp-address |
|---|---|---|
| E1 | Tom | Berkeley |
| E2 | Don | Berkeley |
| E3 | Bob | Princeton |
| E4 | Carol | Berkeley |

### CHILD

| emp-id | child-id | child-name | other-parent-id |
|---|---|---|---|
| E1 | C1 | Jane | — |
| E2 | C1 | Tom | S1 |
| E2 | C2 | Dick | S1 |
| E2 | C3 | Donna | S2 |
| E4 | C1 | Lisa | S1 |

*Sample Instance Tables for EMPLOYEE, CHILD, and SPOUSE*

**SPOUSE**

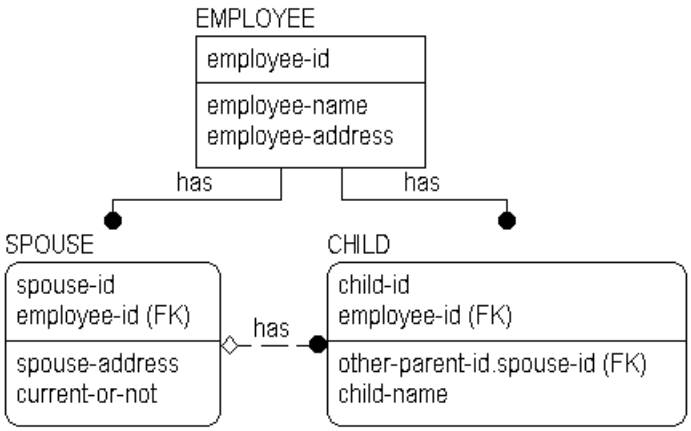
| emp-id | spouse-id | spouse-address | current-or-not |
|---|---|---|---|
| E2 | S1 | Berkeley | Y |
| E2 | S2 | Cleveland | N |
| E3 | S1 | Princeton | Y |
| E4 | S1 | New York | Y |
| E5 | S1 | Berkeley | Y |

*Sample Instance Tables for EMPLOYEE, CHILD, and SPOUSE*

The normalization of this model is not completed, however. In order to complete it you must ensure that you can represent all possible relationships between employees and children, including those in which both parents are employees.

## Unification

In the example below, the “employee-id” attribute migrates to the CHILD entity through two relationships – one with EMPLOYEE and the other with SPOUSE. You might expect that the foreign key attribute would appear twice in the CHILD entity as a result. However, because the attribute “employee-id” was already present in the key area of CHILD, it is not repeated in the entity even though it is part of the key of SPOUSE.




*Unification of the “Employee-id” Foreign Key Attribute*

This combining of two, identical foreign key attributes migrated from the same base attribute through two or more relationships is called ***unification***. In the example, “employee-id” was part of the primary key of CHILD (contributed by the “has” relationship from EMPLOYEE), and was also a non-key attribute of CHILD (contributed by the “has” relationship from SPOUSE). Because both foreign key attributes are the identifiers of the same EMPLOYEE, it is desirable that the attribute appears only once. Unification is implemented automatically by ERwin when this situation occurs.

The rules that ERwin uses to implement unification include:

1. If the same foreign key is contributed to an entity more than once, without the assignment of rolenames, all occurrences unify.
2. Unification does not occur if the occurrences of the foreign key are given different rolenames.
3. If different foreign keys are assigned the same rolename, and these foreign keys are rolenamed back to the same base attribute, then unification will occur. If they are not rolenamed back to the same base attribute, there is an error in the diagram.

4. If any of the foreign keys that unify are part of the primary key of the entity, the unified attribute will remain as part of the primary key.
5. If none of the foreign keys that unify are part of the primary key, the unified attribute will not be part of the primary key.

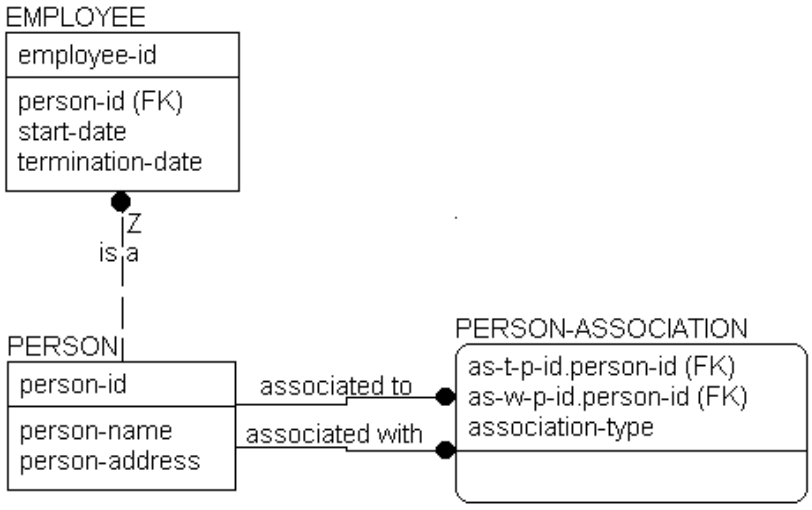
Accordingly, you can override the unification of foreign keys when necessary by assigning rolenames. If you want the same foreign key to appear two or more times in a child entity, you can add a rolename to each foreign key attribute.

## How Much Normalization Is Enough?

From a formal normalization perspective (what an algorithm would find solely from the "shape" of the model, without understanding the meanings of the entities and attributes) there is nothing wrong with the EMPLOYEE-CHILD-SPOUSE model. But just because it is normalized does not mean that the model is complete or correct. It still may not be able to store all of the information that is needed, or may store it inefficiently. With experience, you can learn to detect and remove additional design flaws even after the "pure normalization" is finished.

In the earlier EMPLOYEE-CHILD-SPOUSE model example, you have already discovered that there is no way of recording a CHILD whose parents are both EMPLOYEEs. So you can make additional changes to try to accommodate this type of data.

If you noticed that EMPLOYEE, SPOUSE, and CHILD all represent instances of people, you may want to try to combine the information into a single table that represents facts about people and one that represents facts about relationships. To fix the model, you can eliminate CHILD and SPOUSE, replacing them with PERSON and PERSON-ASSOCIATION. This lets you record parentage and marriage through the relationships between two PERSONs captured in the PERSON-ASSOCIATION entity.

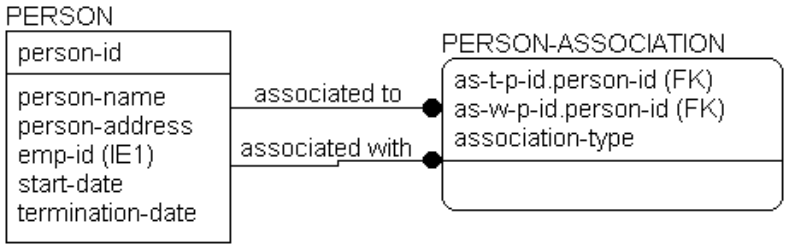



*EMPLOYEE, PERSON, and PERSON-ASSOCIATION Entities*

In this structure, we can finally record any number of relationships between two PERSONs, as well as a number of relationships we could not record before. For example, the previous model had not considered “adoption.” The new structure, however, automatically covers it. To represent “adoption” you can add a new value to the “person-association-type” validation rule to represent adopted parentage. You can also add “legal guardian,” “significant other,” or other relationships between two PERSONs later, if needed.

EMPLOYEE remains an independent entity, because the business chooses to identify EMPLOYEEs differently from PERSONs. However, EMPLOYEE inherits the properties of PERSON by virtue of the “is a” relationship back to PERSON. Notice the “Z” on that relationship and the absence of a diamond. This is a one-to-zero or one relationship that can sometimes be used in place of a subtype when the subtype entities require different keys. In our example, a PERSON either “is an” EMPLOYEE or not.

If we had wanted to use the same key for both PERSON and EMPLOYEE we could have “encased” the EMPLOYEE entity into PERSON and allowed its attributes to be NULL whenever the PERSON was not an EMPLOYEE. We still could have specified that the business wanted to look up “employees” by a separate identifier, but the business statements would have been a bit different. This structure is shown below.




*Generalized Employee Model Construct*

## Conclusions

What this all basically comes down to in the end is that a model may “normalize,” but may still not be a correct representation of the business. Formal normalization is important. Verifying, perhaps with sets of sample instance tables as we have done here, that the model means something is no less important.

## ERwin Support for Normalization

ERwin provides some support for normalization of data models, but does not currently contain a full normalization algorithm. If you have not used a "real time" modeling tool before, you will find ERwin's standard modeling features quite helpful. They will prevent you from making many normalization errors in the first place.

### First Normal Form Support

In a model, each entity or attribute is identified by its "name." ERwin will accept any name for an object, with the following exceptions:

- ERwin will flag a second use of an entity name (depending on your preference for unique names).
- ERwin will flag a second use of an attribute name, unless that name is a rolename. When rolenames are assigned, the same name for an attribute may be used in different entities.
- ERwin will not let you bring a foreign key into an entity more than once without giving it a rolename each time.

By preventing multiple uses of the same name, ERwin is basically leading you to put each fact in exactly one place. There may still be second normal form errors if you place an attribute incorrectly, but no algorithm would find that without more information than is present in a model.

In the data model, ERwin cannot know that a name you assign to an attribute can represent a "list" of things. For example, ERwin was happy to accept "children's-names" as an attribute name in our earlier example. So it does not directly guarantee that every model is in first normal form.

However, the DBMS schema function of ERwin does not support a data type of "list." Because the schema is a representation of the database in a physical relational system, first normal form errors are also prevented at this level.

## Second and Third Normal Form Support

ERwin does not currently know about functional dependencies, but it can help to prevent second and third normal form errors. For example, if you reconstruct the examples presented earlier in this chapter, you will find that once “spouse-address” has been defined as an attribute of SPOUSE, you cannot also define it as an attribute of CHILD. (Again, depending on your preference for unique names.)

By preventing the multiple occurrence of foreign keys without rolenames, ERwin is reminding you to think about what the structure represents. If the same foreign key occurs twice in the same entity, there is a business question to ask:

> ***Are we recording the keys of two separate instances, or do both of the keys represent the same instance?***

When the foreign keys represent different instances, separate rolenames are needed. If the two foreign keys represent the same instance, then there is very likely a normalization error somewhere. A foreign key appearing twice in an entity, without rolenames, is a dead giveaway that there is a redundant relationship structure in the model. When two foreign keys are assigned the same rolename, unification occurs.

# 7 Creating a Physical Model

## What's In This Chapter?

The objective of a physical model is to provide a database administrator with sufficient information to create an efficient physical database. The physical model also provides a context for the definition and recording in the data dictionary of the data elements that form the database, and assists the application team in choosing a physical structure for the programs that will access the data. To ensure that all systems-side needs are met, physical models are often developed jointly by a team representing the data administration, database administration, and application development areas.

When deemed appropriate for the development effort, the model can also provide the basis for comparing the physical database design against the original business information requirements — to demonstrate that the physical database design adequately supports those requirements, to document physical design choices and their implications (e.g., what is satisfied, and what is not), and to identify database extensibility capabilities and constraints.

ERwin provides support for both roles of a physical model: generating the physical database and documenting physical design against the business requirements. For example, you can create a physical model from an ERD, key-based, or fully attributed model simply by changing the view of the model from "Logical Model" to "Physical Model." Each option in the logical model has a corresponding option in the physical model. So each entity becomes a relational table, attributes become columns, and keys become indices.

Once the physical model is created, ERwin can generate all model objects in the correct syntax for the selected target server – directly to the catalog of the target server, or indirectly, as a schema DDL script file.

### Chapter Contents

## Creating a Physical Model

The following table summarizes the relationship between objects in a logical and physical model.

### Summary of Logical and Physical Model Components

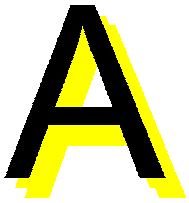
| Logical Model | Physical Model |
|---|---|
| Entity | Table |
| Dependent entity | FK is part of child table's PK |
| Independent entity | Parent table or, if child table, FK is NOT part of child table's PK |
| Attribute | Column |
| Logical datatype (text, number, datetime, blob) | Physical datatype (valid example varies depending on the target server selected) |
| Domain (logical) | Domain (physical) |
| Primary key | Primary key, PK Index |
| Foreign key | Foreign key, FK Index |
| Alternate key (AK) | AK Index—a unique, non-primary index |
| Inversion entry (IE) | IE Index—a non-unique index created to search table information by a non-unique value, such as customer last name. |
| Key group | Index |
| Business rule | Trigger or stored procedure |
| Validation rule | Constraint |
| Relationship | Relationship implemented using FKs |
| Identifying | FK is part of child table's PK (above the line) |
| Non-Identifying | FK is NOT part of child table's PK (below the line) |
| Subtype | Denormalized tables |
| Many-to-many | Associative table |
| Referential Integrity (cascade, restrict, set null, set default) | INSERT, UPDATE, and DELETE Triggers |
| Cardinality | INSERT, UPDATE, and DELETE Triggers |
| N/A | View or view relationship |
| N/A | Pre- and post-script |

***Note**: Referential integrity is described as a part of the logical model, because the decision of how you want a relationship to be maintained is a business decision, but it is also a physical model component, because triggers or declarative statements appear in the schema. ERwin supports referential integrity as a part of both the logical and physical model.*

## Denormalization

ERwin also lets you denormalize the structure of the logical model so that you can build a related physical model that is designed effectively for the target RDBMS. Features supporting denormalization include:

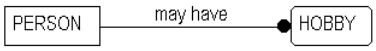
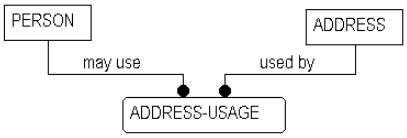
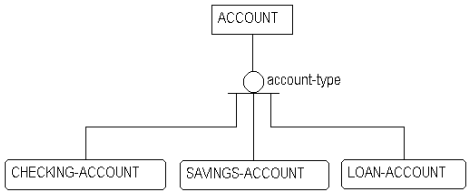
- "Logical only" properties for entities, attributes, key groups, and domains. You can mark any item in the logical model "logical only" so that it appears in the logical model, but does not appear in the physical model. For example, you can use the "logical only" settings to denormalize subtype relationships or support partial key migration in the physical model.
- "Physical only" properties for tables, columns, indexes, and domains. You can mark any item in the physical model "physical only" so that it appears in the physical model only. This setting also support denormalization of the physical model because it enables the modeler to include tables, columns, and indexes in the physical model that directly support physical implementation requirements.
- Resolution of many-to-many relationships in a physical model. ERwin provides support for resolving many-to-many relationships in both the logical and physical models. If you resolve the many-to-many relationship in the logical model, ERwin creates the associative entity and lets you add additional attributes. If you choose to keep the many-to-many relationship in the logical model, you can still resolve the relationship in the physical model. ERwin maintains the link between the original logical design and the new physical design, so the origin of the associative table is documented in the model.



# Dependent Entity Types

## Classification of Dependent Entities

The following table lists the types of dependent entities that may appear in an IDEF1X diagram.

| Dependent Entity Type | Description | Example |
|---|---|---|
| **Characteristic** | A characteristic entity represents a group of attributes which occurs multiple times for an entity, and which is not directly identified by any other entity. In the example, HOBBY is said to be a characteristic of PERSON. |  |
| **Associative or Designative** | Associative and designative entities record multiple relationships between two or more entities. If the entity carries only the relationship information, it is termed a designative entity. If it also carries attributes that further describe the relationship, it is called associative. In the example, ADDRESS-USAGE is an associative or designative entity. |  |
| **Subtype** | Subtype entities are the dependent entities in a subtype relationship. In the example, CHECKING-ACCOUNT, SAVINGS-ACCOUNT, and LOAN-ACCOUNT are subtype entities. |  |

# Glossary of Terms

## Alternate Key

1) An attribute or attributes that uniquely identify an instance of an entity.

2) If more than one attribute or group of attributes satisfies rule 1, the alternate keys are those attributes or groups of attributes not selected as the primary key.

ERwin will generate a unique index for each alternate key.

## Attribute

An attribute represents a type of characteristic or property associated with a set of real or abstract things (people, places, events, etc.).

## Basename

The original name of a rolenamed foreign key.

## Binary Relationship

A relationship in which exactly one instance of the parent is related to zero, one, or more instances of a child. In IDEF1X, identifying, non-identifying, and subtype relationships are all binary relationships.

## Cardinality

The ratio of instances of a parent to instances of a child. In IDEF1X, the cardinality of binary relationships is 1:*n*, whereby *n* may be one of the following:

Zero, one, or more - signified by a blank space

One or more - signified by the letter P

Zero or one - signified by the letter Z

Exactly *n* - where *n* is some number

## Complete Subtype Cluster

If the subtype cluster includes all of the possible subtypes (every instance of the generic parent is associated with one subtype), then the subtype cluster is complete. For example, every EMPLOYEE is either male or female, and therefore the subtype cluster of MALE-EMPLOYEE and FEMALE-EMPLOYEE is a complete subtype cluster.

## Dependent Entity

An entity whose instances cannot be uniquely identified without determining its relationship to another entity or entities.

## Discriminator

The value of an attribute in an instance of the generic parent determines to which of the possible subtypes that instance belongs. This attribute is known as the discriminator. For example, the value in the attribute Sex in an instance of EMPLOYEE determines to which particular subtype (MALE-EMPLOYEE or FEMALE-EMPLOYEE) that instance belongs.

## Entity

An entity represents a set of real or abstract things (people, places, events, etc.) which have common attributes or characteristics. Entities may be either independent, or dependent.

## Foreign Key

An attribute that has migrated through a relationship from a parent entity to a child entity. A foreign key represents a secondary reference to a single set of value values - the primary reference being the owned attribute.

## Identifying Relationship

A relationship whereby an instance of the child entity is identified through its association with a parent entity. The primary key attributes of the parent entity become primary key attributes of the child.

## Incomplete Subtype Cluster

If the subtype cluster does not include all of the possible subtypes (every instance of the generic parent is not associated with one subtype), then the subtype cluster is incomplete. For example, if some employees are commissioned, a subtype cluster of SALARIED-EMPLOYEE and PART-TIME EMPLOYEE would be incomplete.

## Independent Entity

An entity whose instances can be uniquely identified without determining its relationship to another entity.

## Inversion Entry

An attribute or attributes that do not uniquely identify an instance of an entity, but are often used to access instances of entities. ERwin will generate a non-unique index for each inversion entry.

## Non-key attribute

Any attribute that is not part of the entity's primary key. Non-key attributes may be part of an inversion entry and / or alternate key, and may also be foreign keys.

## Non-Identifying Relationship

A relationship whereby an instance of the child entity is not identified through its association with a parent entity. The primary key attributes of the parent entity become non-key attributes of the child.

## Nonspecific Relationship

Both parent-child connection and subtype relationships are considered to be specific relationships because they define precisely how instances of one entity relate to instances of another. However, in the initial development of a model, it is often helpful to identify "non-specific relationships" between two entities. A nonspecific relationship, also referred to as a "many-to-many relationship," is an association between two entities in which each instance of the first entity is associated with zero, one, or many instances of the second entity and each instance of the second entity is associated with zero, one, or many instances of the first entity.

## Primary Key

An attribute or attributes that uniquely identify an instance of an entity. If more than one attribute or group of attributes can uniquely identify each instance, the primary key is chosen from this list of candidates based on its perceived value to the business as an identifier. Ideally, primary keys should not change over time, and should be as small as possible. ERwin will generate a unique index for each primary key.

## Referential Integrity

The assertion that the foreign key values in an instance of a child entity have corresponding values in a parent entity.

## Rolename

A new name for a foreign key. A rolename is used to indicate that the set of value values of the foreign key is a subset of the set of value values of the attribute in the parent, and performs a specific function (or role) in the entity.

## Schema

The structure of a database. Usually refers to the DDL (data definition language) script file. DDL consists of CREATE TABLE, CREATE INDEX, and other statements.

## Specific Relationship

A specific relationship is an association between entities in which each instance of the parent entity is associated with zero, one, or many instances of the child entity, and each instance of the child entity is associated with zero or one instance of the parent entity.

## Subtype Entity

In the real world, we often encounter entities which are specific types of other entities. For example, a SALARIED EMPLOYEE is a specific type of EMPLOYEE. Subtype entities are useful for storing information that only applies to a specific subtype. They are also useful for expressing relationships that are only valid for that specific subtype, such as the fact that a SALARIED EMPLOYEE will qualify for a certain BENEFIT, while a PART-TIME-EMPLOYEE will not. In IDEF1X, subtypes within a subtype cluster are mutually exclusive.

## Subtype Relationship

A subtype relationship (also known as a categorization relationship) is a relationship between a subtype entity and its generic parent. A subtype relationship always relates one instance of a generic parent with zero or one instance of the subtype.

# Index

# Documentation Comments Form

**ERwin Version 3.0**
**Methods Guide**

Logic Works is interested in your feedback on this documentation. You can use this form if you have compliments or questions, or would like to report problems in the documentation.

Please fax or mail completed forms to:

*Documentation Manager*
*Logic Works, Inc.*
*University Square at Princeton*
*111 Campus Drive*
*Princeton, NJ 08540*

*Fax: (609) 514-0184*

## Comments:

Please enter your comments in the space provided below:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please include your name, address, and telephone number in the space below:

_____

_____

_____

Would you like a Logic Works representative to contact you? Yes ☐ No ☐