Introduction

- Transport databases
- Data models
- Agile methods
- Building the agile geodatabase
- Book organization

Modes of travel can be quite different but all follow a conceptual structure consisting of an origin, a destination, a path between the two, and a conveyance to move along the path. *Designing Geodatabases for Transportation* tells how to design a geospatial information system to manage data about transportation facilities and services.

An enterprise geodatabase can help solve two common transportation challenges: the many origins, destinations, paths, and conveyances that may be present; and the need to specify locations along the facility. There is also the matter of facility suppliers usually being different from facility users. The facility user focuses on origins and destinations. The facility supplier is concerned
about the many paths over which a conveyance travels and generally not about specific trips. A transportation agency provides the facilities to support travel. Shippers—those who have goods to deliver—define the origin and destination for their shipment. A shipping company supplies the conveyances and selects the paths to move the shipper’s goods from origin to destination. Each origin and destination must be accessible from a transport facility. A shipping company can select a path only where a corresponding facility exists. Railroads perhaps can be viewed as being both suppliers and users of transport capacity although some may operate over facilities they do not own.

**Transport databases**

Transportation data is often specific to the various modes of transport. *Designing Geodatabases for Transportation* addresses six modes: walking, bicycling, motor highways, public transit (buses and commuter rail), railroads, and navigable waterways. All six modes involve linear facilities supporting point-to-point travel for people and material goods. The nature of the facility supporting travel and the way it is used differs with each mode. Some facilities support multiple modes of travel. Highways and roads accommodate motor vehicles, pedestrians, and bicycles. Railroads support commuter trains, long-distance passenger travel, and freight movement. Ships travel navigable waterways that flow under highway and railroad bridges.

Points of modal connection are commonly known as terminals, depots, stations, stops, and crossings. The name “intersection” is usually applied to highways, but conceptually includes other points where facilities cross and interact, such as railroad switches, crossovers, and diamonds; rail-highway grade crossings; and limited-access highway interchanges. Transport systems also include places where facilities cross but do not intersect, such as at bridges, viaducts, and tunnels.

Geographic information systems (GIS) for transportation—GIS-T in industry shorthand—routinely deal with mode- and function-specific applications, each with its own geodatabase design. What is rare is a GIS-T geodatabase that goes beyond serving the needs of a single application. Such a geodatabase must accommodate the many segmentation schemes employed and the various linear and coordinate referencing systems available to show where the elements, conveyances, and characteristics of transportation systems are located. *Designing Geodatabases for Transportation* shows you how to construct such an enterprise multimodal geodatabase, although the ideas presented in this book can be implemented for a single mode.

A transportation geodatabase addresses concerns beyond facilities and the services that use them. For example, facility elements and characteristics are affected by projects and
activities that construct, maintain, and remove elements of the transportation system. There are also traffic crashes, bus routes, train schedules, and shipping manifests to consider.

Transportation applications are much too diverse for this book to present you with a complete transportation database design. That task is up to you. ESRI has successfully worked with user groups to develop a number of industry-specific data models. That approach will not work with transportation, which lacks a single, all-encompassing view of the industry due to its diversity. Not only are there modal differences to consider, but there are also differences in detail and abstraction. A trucking company, a city public-works division, and a state department of transportation (DOT) all need data about highways, but for their own purposes that require them to adopt very different data models. Even within a single transport agency there may be several different application-specific data models.

What this book offers is a collection of ideas and geodatabase design components to help you construct a model that serves your agency and its unique set of applications. It shows you a variety of ways to handle a specific data need, describing the pros and cons of each choice. In this way, *Designing Geodatabases for Transportation* provides a cafeteria of design options rather than a fixed menu to solve the broad range of transportation spatial data requirements.

**Data models**

Geodatabase design is normally expressed through a data model, which is a graphical way of describing a database. A data model is essentially a set of construction plans for a database. Some data models are very conceptual, others extremely detailed. Fortunately, all data models use a few very simple symbols. You need no prior experience with data models or geodatabase design to understand and apply the suggestions in this book.

All geodatabases form a set of abstract representations of things in the real world. The process of abstraction is called modeling. You may geometrically represent linear transport facilities in your abstract geodatabase world as lines. In the real world, of course, transport facilities are areas, which may be less abstractly represented by polygons. Unfortunately, many of the analytical techniques you will want to employ, such as pathfinding, do not exist in the polygon world. This is not really a problem, though, because the central aspect of a linear transport facility is its linearity, so points and lines form the abstract world of most transport geodatabases.

Whether you formally draft a data model or not, one exists inside each geodatabase. It may also exist externally as a set of requirements, a list of class properties, or some other description of the geodatabase's contents and structure. ESRI's ArcGIS software comes with tools to produce a data model from an existing geodatabase. As geodatabases grow from
supporting isolated workgroups to major portions of a complete enterprise, the complexity of geodatabase design normally increases in proportion to the number of data uses. Explicit data models and other documentation become more important as the scope of a geodatabase expands or it begins to serve a critical function within the organization. As a result, many organizations have invested significant resources into developing a good data model before moving forward with geodatabase development and migration projects. More comprehensive and ambitious projects seek to deploy at the enterprise level.

Building a good enterprise geodatabase starts with an enterprise data model. Accordingly, this book expands and enhances the previous ESRI transportation data model, UNETRANS (Unified Network for Transportation), developed by the University of California at Santa Barbara (UCSB) with financial support from ESRI. The “new and improved” UNETRANS is designed to provide a full structure that embraces all of a large organization’s data and access mechanisms. This enterprise data model also takes advantage of technological advances in the ArcGIS family since the original UNETRANS was developed several years ago.

However, developing an enterprise data model need not be the first step in creating an enterprise geodatabase. Your organization may first want to construct a workgroup prototype to gain experience with the geodatabase structure and how to use it. Effective geodatabase design and deployment is an ongoing process. The needs of the organization and the capabilities of the technology both evolve. You will need an enterprise architecture that describes the computing environment and its business rules. Everything else will be somewhat reactive because you cannot possibly anticipate all data uses. You must be agile.

**Agile methods**

Since modern geodatabase design should be based on data modeling, this book includes a review of that process in chapter 2. But it is important to note that data modeling fits within a larger information technology process. Years ago, data modeling and application development were separate processes. Now, it is generally recognized that the speed of database access determines, to a large degree, application performance. This change is partly the result of application intelligence being built into the database management system, and partly the product of larger datasets with more data types. As a result of the stronger role for databases in application performance, data modeling is now central to application development with both following a common design process.

The agile data modeling and application design approach has been described by many practitioners, such as Scott W. Ambler, a Canadian consultant who has written several books on the subject. It is the key to surviving the changing needs of an organization and the potential solutions offered by technology. Attempting to define all the requirements up front
is much riskier than following an agile methodology that can accommodate change. Thus, *Designing Geodatabases for Transportation* presents an enterprise data model only to demonstrate how all the pieces may be put together, not because the enterprise data model is a necessary first step to implementation. Again, the solutions offered here are more like the choices in a cafeteria: you can pick the ones you want. This book also guides you on making your selections. One size does not fit all.

The agile method uses an iterative, team-based approach. Members of the team are not given separate assignments; all team members work together on each assignment. The ideal team member will have broad knowledge of the organization and its operational needs, the general requirements of the agile method, and expertise in one or more areas required for design development and implementation. The hallmarks of the agile method are its use of “good enough” documentation, which includes data models; frequent deliveries of solution components to get user feedback; and structured performance testing at the planned scale of deployment (number of users, transaction types and volumes, etc.). The number of and execution time for database queries and related operations determine geodatabase performance and, thus, application performance. While there may be several ways to reach the same result, you will want to use the one that provides the greatest performance. No one likes to wait. The only way to know which approach provides suitable performance is to

![Geodatabase design process](image-url)
test it under real-world conditions. Performance testing must be part of the geodatabase design process.

Figure 1.2 Agile design process at the enterprise level This figure shows the agile design process with extra structure when applied to an entire enterprise. The box on the right includes four of the eight components shown in figure 1.1. The four boxes on the left side provide the enterprise context for the agile process. High-level requirements define the overall scope. The system architecture defines the deployment environment, such as which relational database management system (RDBMS) product will be used, who will have responsibility for system maintenance, where the hardware will be located, and other constraints. Proof-of-concept prototypes may be necessary to ensure that the system architecture will perform as expected under production loads.

When you seek to apply the agile method to an entire enterprise, you need to break the project into deliverable components that fit within an overall framework. The enterprise framework allows the organization to balance the needs of various internal and external communities competing for limited development resources and to test fundamental aspects of the overall system architecture vital to major parts of the final system.

It has not been assumed that the enterprise of interest is a transport agency. This book takes into account the fact that many transportation data users do not actually care that much about transportation per se. These users need transportation data as a reference plane for other information, such as situs addresses tied to relative position along a transportation facility (street address). Map making is generally placed in this category. Thus, this book
will show how to build transportation-oriented geodatabases that can support the editing of data needed by a variety of applications. The focus will not be solely on the more extensive data models suitable for larger organizations. The book starts with simple problems and solutions, growing in complexity in response to application requirements until it reaches the most complex problems posed by large transport agencies and transit operators. Some chapters, such as those on navigable waterways and railroads, are actually directed to users outside these industries.

*Designing Geodatabases for Transportation* also does not assume you are starting from scratch. The more common starting point today is the migration of a transportation dataset based on the coverage or shapefile structure to a geodatabase, often in the midst of making other changes. A common concurrent technology change is adopting a relational database management system (RDBMS) for overall data storage. Indeed, the migration to the geodatabase structure often marks the point where spatial data goes from being concentrated in workgroup datasets to becoming an enterprise resource available to the entire organization. In any event, you probably are not starting with a blank sheet of paper but rather are dealing with a number of prior decisions you need to identify in the requirements analysis and system architecture steps. You will rely on your agility to work within an existing data structure serving a number of existing applications and within a prescribed set of constraints and resources.

**Building the agile geodatabase**

Your task will be made much easier if you build an agile geodatabase. This starts by separating the editing geodatabase environment from the published dataset. The purpose of any GIS application is to create information. You have to put data into an application in order to get information out. No single book can show you the answer to every geodatabase design problem given the vast range in possible applications, even if the scope is restricted to a single data theme. So, instead of trying to solve all transportation application problems, *Designing Geodatabases for Transportation* seeks to solve only one: creating information out of original data sources, which is data editing. The data thus maintained is then used to populate datasets supporting other applications, each of which presents its own data structure and content requirements. The outputs of the editing process are defined by all the other applications.

Many of the applications’ data requirements are likely to suggest geodatabase design characteristics that work against the data-editing process. What you really need to do is view data editing as its own application, one that creates the inputs to all the other applications, and then follows the mantra that the application determines the geodatabase design.
For example, eliminating data redundancies greatly benefits data editing by keeping each piece of information in a single location so you do not have to enter it multiple times. Thus, a geodatabase optimized for editing will eliminate data redundancies that cause extra work and increase the chance for inconsistencies.

Right about now, you may be thinking that you never want data redundancies, so why is this a big deal? The answer is that no single application may impose the need for data redundancy, but the collection of applications supported by the editing process may. For example, you could have several applications that want to know the length of a facility, some in meters and others in miles. Even if all the applications want the data in the same form, they are likely to expect the data to be stored in a data field under a specific name, such as LEN, LENGTH, or DISTANCE. Rather than store the length in all these different forms and field names within the editing database, you want to store it there once and then create the different versions needed by the various supported applications. There may also be applications that need data derived from other data. For instance, sums, averages, minimums, maximums, and counts may be employed by various applications, such as the total number of highway lane miles or the minimum length of all passing sidings located along a rail line. It is much better to have these values derived rather than enter them directly because it saves time and reduces the risk of error.

These practices mean the process of moving data from the editing to the published geodatabase will likely involve data transformations, calculation of derived fields, data replication, and other actions. But this process can be automated. In contrast, data editing is a primarily manual task. Work smarter, not harder. You will get better data with less work.

Going back to the earlier discussion of agile methods in enterprise geodatabase design, the editing environment is typically the last one to be designed and the first to be built. It is designed last because you will not know what data must be maintained—and the geodatabase design that best supports that data—until all the application inputs are defined. It is delivered first because all those using applications will not function until the inputs are provided. Assuming you cannot design and build everything at once, this chronology presents an impossible task, because the agile method assumes that an application’s final requirements evolve. As a result, the geodatabase must itself be agile.

The core concept of agility is flexibility combined with robustness. Separating the editing and usage portions of the complete enterprise database allows each to evolve independently and to use a structure optimally suited to its needs. Editing involves lots of small transactions that change the geodatabase coupled with a strong need to coordinate edits made by different persons over time. In other words, maintaining database integrity. In contrast, applications involve extractions of relatively large chunks of data. Each application defines a set of data needs and imposes requirements on the geodatabase that it uses. That geodatabase should be part of the published dataset, which receives its content from the editing
geodatabase. If you use the editing geodatabase directly, then your application would have to do all the heavy lifting associated with getting the data into the right form. Conversely, if you edit the application’s geodatabase directly, then the editing process has to deal with the data structure the application needs. In both cases, you have editors and users churning the same data, which can often produce surprising results because of a loss of referential integrity; i.e., differences in values across the geodatabase.

This book provides detailed instructions for how to structure and process data so that it can be used to support applications without users having to separately and duplicatively maintain the data. This book is about enterprise data editing, not within a single office, but across the organization. The data it embraces is defined by other applications. Editing geodatabases evolve more frequently than do application geodatabases. The editing environment is the sum of all application data requirements. As a result, it will probably need to be modified each time any application changes or is added to the list of supported work processes.

*Designing Geodatabases for Transportation* describes a geodatabase design process founded on content rather than specific applications. The design of the editing application is determined by the nature of the data to be edited. Thus, the solutions presented in this book follow the general structure of, “If your user needs this kind of data, then build the editing geodatabase this way.” Many of the geodatabase design principles presented here are widely applicable and need not be restricted to transportation themes. All are consistent with good data-management practices and current technology.

### Book organization

This book is divided into three parts. Part 1 covers the basics of geodatabase design. Part 2 explores the various ways transportation geodatabases may be structured. Part 3 offers a variety of advanced topics on transportation geodatabase design.

As with any book intended for a wide range of readers, *Designing Geodatabases for Transportation* covers a lot of foundational concepts dealing with database design in general and geodatabases in particular. While it may tempting for a more knowledgeable reader to skip the first few chapters, even the advanced data modeler should review the content of part 1 in order to be familiar with the terms and presentation employed in this book. Similarly, you may want to explore the modal chapters in part 3 related to forms of transportation not included in your own geodatabase because there may be ideas you can use.

One of the more obvious demarcations in the book is the distinction between the segmented data structures used mainly by local governments and commercial database vendors and the route-based structures used primarily by state and provincial transport agencies. Because
they are conceptually less complex, design concepts more applicable to segmented data models are generally presented in earlier chapters and those concepts with greater applicability to route-based models are covered in later chapters. Do not skip the content directed to one side of this dividing line because this distinction in application is often one of convenience. Many design techniques are applicable to both basic data structures.

Some content is targeted to a specific audience. These passages will be placed in sidebars identified by one of two icons.

The building block icon identifies basic knowledge about a fundamental aspect of the topic.

The rocket icon denotes information suitable for advanced readers that describes what is happening behind the scenes, gets into the details of a topic, or offers guidance for specific tasks.

Transportation geodatabases have been difficult to construct in the past, in large part because of a lack of basic guidance on how to address the many problems presented by this unique data and the business processes it supports. *Designing Geodatabases for Transportation* is intended to provide basic guidance on how to construct transportation geodatabases in a manner that addresses these inherent problems.

Notes

1 Although pipelines and utility systems have a similar structure and do transport materials or energy from place to place, they are not included in this book. Other ESRI Press publications and data models address the spatial database design needs of pipelines, telecommunications, water utilities, and electric power systems.
Data modeling

- Data types
- Files
- Tables
- Relationships in relational databases
- Object-relational databases
- Relationships in object-relational databases
- The data-modeling process
- Conceptual data models
- Logical data models
- Physical data models
This chapter covers data modeling, the process of designing a dataset’s structure by adopting a set of abstractions representing the real world. A dataset is a collection of facts organized around entities. An entity is a group of similar things, each of which may be referred to as an instance or a member. For example, Road could be an entity representing all roads, with State Route 50, Interstate 10, Main Street, and Simpson Highway members of that entity. You cannot store the real-world entity in the dataset, so you store a set of descriptive attributes that allow you to identify the entity and understand its characteristics. Attributes can be composed of text, numbers, geometry, images, and other forms of data. If Road is your entity, then facility ID, route number, street name, length, jurisdiction, and pavement condition could be useful attributes. When an attribute involves location, it is considered to be spatial in nature. GIS involves spatial data. Attributes, not entities, determine whether a dataset is spatial.

A database is a dataset stored in an electronic medium. A geodatabase includes spatial data. A user acts upon such a dataset through a database management system, which may also provide various security and data integrity services. A geodatabase is a collection of geographic datasets. The database management systems used for large workgroup and enterprise geodatabases are relational, which means they perform according to a number of rules, called relational algebra, that describe how to read and write information stored in the database. The language shared by relational database management system (RDBMS) products is SQL, which once stood for Structured Query Language. The RDBMS converts SQL statements entered by the user (or generated by a computer application) into relational algebra to perform operations on the data. You do not need to know about RDBMS products, relational algebra, or SQL to do data modeling. What you do need to know is included in this chapter.

Every database is a data model because a model is simply an abstract representation of the real world. A primary concern of data modeling is deciding which abstraction to use. For example, a spatial database may represent a linear transportation facility with a centerline, but the real-world facility is actually an area with one very long axis. We commonly use a centerline because it conveys the primary aspect of the facility: it has length and traverses a space. That centerline can be part of a geometric network for determining the best path between two points, or it can simply be a reference for locating other features on a map. The information you need about the facility is determined by how the data will be used. The network pathfinding application will need information about connectivity, cost of traveling on a segment, and restrictions to travel. Any geometric representation you created for the network may be highly abstract, perhaps just a straight line between two points. In contrast, a mapping application needs just a line geometry representation, with perhaps some information for symbolizing the line. The scale of display will determine the degree of abstraction allowed for the geometry. Large-scale maps may need detailed road edgelines, while small-scale maps may need only a generalized centerline.
You can alternatively represent the linear facility as a surface, such as might be done for a digital elevation model (DEM) using a triangulated irregular network (TIN), which are both ways to represent a surface for 3D representations, or it could be a set of pixels in a raster image. You might also store the linear facility as a set of address points. You can even store the facility as a set of nonspatial attributes, employing no geometry at all. Each of these abstractions has a place within a transport agency and its variety of spatial-data applications. However, this book will concentrate on vector data forms where lines represent linear facilities, as this is the most common abstraction. Several design proposals show how to accommodate multiple geometric representations for a single entity.

Your choice of which form of abstraction to use is determined by the data’s application. Since larger transportation organizations need many applications, it is likely they will need multiple abstractions. For example, a bridge might be a point feature to some, a linear feature to others, and a polygon feature to yet another group.

Data modeling is the structured process by which you examine the needs of your application and determine the most appropriate abstraction to use. It begins by understanding the application’s requirements for data, which will determine the appropriate level of abstraction, the structure to use in organizing the data, the entities to be created, and the attributes assigned to each entity. In the geodatabase, an entity eventually becomes a class, which is a discrete table or feature that you will define in terms of its properties, behaviors, and attributes. A geodatabase combines data with software in an object-oriented form that takes over much of the workload needed to use and manage the data. A geodatabase is an active part of the ArcGIS platform, not a passive holder of data.

Much more about these concepts is discussed later in this chapter. What you need to know for now is that data modeling, as presented here, is founded on the capabilities and constraints of the geodatabase. However, if you are like most transportation-data users, you already have data in a variety of nongeodatabase forms, so this chapter covers other fundamental data structures along with the basic concepts of database design and data models.

**Data types**

When starting a data-modeling project, you first must understand the data you intend to place into your new geodatabase. In addition to the geometry you use to abstractly represent a real-world entity cartographically, you have the traditional forms of data that have always been part of transport databases.
Chapter 2: Data modeling

One of the most common kinds of data is text, which consists of a string of alphanumeric characters, such as letters, numbers, and punctuation. You specify the number of characters as part of the data type definition when you add an attribute of this type to a geodatabase class.

Numbers
The four numeric data types normally used in geodatabases may be defined using by specifying type, precision, and scale. Precision is the total number of digits accommodated. Scale is the maximum of digits permitted after the decimal. The actual numerical range limits for precision and scale are somewhat dependent on the RDBMS you use. Precision and scale specifications are used only when creating an ArcSDE geodatabase; only a data type choice is needed for files and personal geodatabase classes.

Date and time
The Date data type includes both date and time information presented as the date first, and then the time, with a resolution of thousandths of a second. Entries with only date information will have zeroes placed in the time components. Similarly, time-only entries will have zeroes in the date components.

Character strings
A character string is any value consisting of printable alphanumeric characters, such as letters, numbers, and punctuation. You specify the number of characters as part of the data type definition when you add an attribute of this type to a geodatabase class.

Numbers
The four numeric data types normally used in geodatabases may be defined using by specifying type, precision, and scale. Precision is the total number of digits accommodated. Scale is the maximum of digits permitted after the decimal. The actual numerical range limits for precision and scale are somewhat dependent on the RDBMS you use. Precision and scale specifications are used only when creating an ArcSDE geodatabase; only a data type choice is needed for files and personal geodatabase classes.

Date and time
The Date data type includes both date and time information presented as the date first, and then the time, with a resolution of thousandths of a second. Entries with only date information will have zeroes placed in the time components. Similarly, time-only entries will have zeroes in the date components.

Figure 2.1  Data types  The ArcGIS geodatabase supports several data types for user-supplied class attributes. The primary ones you will likely use are character string, short integer, long integer, single-precision floating-point (float), double-precision floating-point (double), and date.

One of the most common kinds of data is text, which consists of a string of alphanumeric characters, like letters, numbers, and punctuation. Anything you can type on a keyboard can go into a text field. The maximum number of allowable characters defines most text fields. For example, you might see a reference like “String (30)” to define a text field with a maximum length of 30 characters.

An equally popular form of data is a number. There are many different types of number data, but to the user they all consist of a series of digits. Where they differ is how they are stored in the database. In the geodatabase, a short integer will be stored using 2 bytes of
memory; a long integer requires 4 bytes. A single-precision or floating-point number is also stored using 4 bytes, while a double-precision number is stored using 8 bytes. The actual numeric range that each of these forms represents varies according to the database management system you use.

Working in concert with the type of number format you select is the way you specify it in an ArcSDE geodatabase. A number field in such a database has two characteristics that go with its type. The first is precision, which specifies the maximum number of digits that can be stored. The second characteristic is scale, which tells the database how many of those digits will fall after the decimal point.

Number type, precision, and scale interact in various ways. For example, the database will ignore scale if you specify a number type of integer, because integers consist only of whole numbers. The data type overrides the specification. Sometimes it works the other way. For instance, if you specify a floating-number (single-precision) data type but a precision of seven or more, ArcGIS will change the data type to double-precision.

Most database management systems also support date and time data types. Although stored in vendor-specific ways, ArcGIS provides a consistent representation to the user in which date and time are combined into one data type, called ‘Date’. The date portion is provided as a two-digit month, a two-digit day, and a four-digit year, with the three components separated by a forward slash character. The time portion is presented as a two-digit number representing a 24-hour clock (00-23), a two-digit minute portion (00-59), and a second component with a precision of 5 and a scale of 3. The three time components are separated using colons.

Files

Relational databases were not the first kind of electronic data structure. The oldest form of database storage is the file, which consists of a block of data organized into logical groups called fields. Each position in the field is called a column. Files look like a table with their records (rows) that separate content using a special character to signify the end of a logical group of data. Everything is text. There is no inherent requirement for all the records to have the same structure. For example, the first record, often called a header, could state the number of body records or describe the fields in those records. All the intelligence needed to understand the file’s content is in the application that reads and writes records.
Files come in two basic forms: fixed-length and variable-length. A fixed-length file uses the position of each character in a record to interpret its meaning. Any leftover space not needed to store the data for that record is filled with spaces, either before or after the actual data in the field. Fields in each record are identified by position. For example, a file specification may declare that record characters (columns) 1 through 47 contain an employee’s name right-justified with leading spaces.

**Fixed-length Records**

<table>
<thead>
<tr>
<th>Record</th>
<th>Employee ID</th>
<th>Number of Dependents</th>
<th>Present Age</th>
<th>Last Name</th>
<th>Position Number</th>
<th>End of Record Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 0 7 3 2 9</td>
<td>BUTLER</td>
<td>7 3 2 9</td>
<td>BUTLER</td>
<td>S M I T H 4 2 0 4</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>4 6 2 9 7 4 5</td>
<td>S M I T H</td>
<td>4 2 0 4</td>
<td>S M I T H</td>
<td>J O N E S 3 3 4 1</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>1 8 4 2 0 7 0</td>
<td>J O N E S</td>
<td>3 3 4 1</td>
<td>J O N E S</td>
<td>W I L S O N 7 6 2 6</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>3 0 9 2 1 2 8</td>
<td>W I L S O N</td>
<td>7 6 2 6</td>
<td>W I L S O N</td>
<td>W A S H I N G T O N 6 1 2 4</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>1 9 7 0 2 5 6</td>
<td>W A S H I N G T O N</td>
<td>6 1 2 4</td>
<td>W A S H I N G T O N</td>
<td>B R O W N 1 7 9 9</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>3 2 0 1 0 3 5</td>
<td>B R O W N</td>
<td>1 7 9 9</td>
<td>B R O W N</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
<td>$^\text{13}$</td>
</tr>
</tbody>
</table>

**Variable-length Records**

<table>
<thead>
<tr>
<th>Record</th>
<th>Employee ID, Number of Dependents, Present Age, Last Name, Position Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 0 7 , 3 , 2 9 , BUTLER , 7 3 2 9</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>4 6 2 9 , 7 , 4 5 , S M I T H , 4 2 0 4</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>1 8 4 2 , 0 , 7 0 , J O N E S , 3 3 4 1</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>3 0 9 2 , 1 , 2 8 , W I L S O N , 7 6 2 6</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>1 9 7 0 , 2 , 5 6 , W A S H I N G T O N , 6 1 2 4</td>
<td>$^\text{13}$</td>
</tr>
<tr>
<td>3 2 0 1 , 0 , 3 5 , B R O W N , 1 7 9 9</td>
<td>$^\text{13}$</td>
</tr>
</tbody>
</table>

**Figure 2.2 Files** A fixed-length file uses column position to identify specific data content forming attributes. A variable-length file uses the sequential order of fields separated by a predefined special character—one that cannot appear in the data. In both cases, the application using the data must know the specific location of each piece of information.

Files come in two basic forms: fixed-length and variable-length. A fixed-length file uses the position of each character in a record to interpret its meaning. Any leftover space not needed to store the data for that record is filled with spaces, either before or after the actual data in the field. Fields in each record are identified by position. For example, a file specification may declare that record characters (columns) 1 through 47 contain an employee’s name right-justified with leading spaces.
A variable-length file uses the position of a field within the record to identify its content. Variable-length records avoid space filling by using special characters to say where one field stops and another begins. You may have come across this structure when using comma- and tab-delimited text files. The commas or tab characters are the things that separate the records into fields. Usually, there is also a special end-of-file character.

The most common ArcGIS file-based data structure is the shapefile. A shapefile is a kind of spatial database structure consisting of several files. There are more than a hundred recognized shapefile component types, each with its own file extension (the three characters after the dot in a typical file name). To copy a shapefile, you must copy all the component files. The minimum components are the geometry (.shp), the nonspatial attribute data (.dbf), and the spatial index (.sbx). The structure of each component file is optimized for the information it contains. For example, the geometry file (.shp) contains a 100-byte fixed-length file header followed by variable-length records. The variable-length record is composed of an 8-byte, fixed-length record header followed by variable-length record contents. Each record defines a single geometry, with the length of the variable portion being determined by the number of vertices and whether measure (m) and elevation (z) coordinate values are included. The fixed-length record header portion provides a record number and the length of the variable portion.

Coverages, which were the original ESRI data structure, are also based on a database structure consisting of multiple files. Designed to reduce the size of a spatial database, software manipulating coverage data must manage a number of composition relationships inherent in the file structure. A special data-exchange file type was developed to be able to distribute coverages via a single file.

File data structures remain useful today and will continue to be part of GIS datasets long into the future. This book, however, will restrict itself to modeling geodatabases. What you put into and take out of a geodatabase may be a file, but the database to be modeled is a geodatabase.

Tables

The next step along the evolutionary line of database design is the table, which is a fundamental data organization unit of a relational database. A table looks very much like a file in the way it is presented to the user by the RDBMS in which it exists: a set of rows (records) and columns (fields). Rows represent members and columns represent attributes. However, you cannot simply copy a table as you can a file, because each table is tightly bound to the RDBMS.
Dr. Edgar Codd invented relational databases in the early 1970s at IBM, although it was several years later before a working product could be devised. Such a database management system is based on relational algebra, a kind of math that controls what can happen to data in such a storage structure. Relational algebra supports seven functions:

- Retrieve (read) row
- Update (write) row
- Define virtual relations (table views)
- Create a snapshot relation
- Define and implement security rules
- Establish and meet stability requirements
- Operate under integrity rules

Figure 2.3 Tables  An application seeking to use the data stored in a relational table needs to know the name of the table and the name of each attribute it seeks, but not the physical manner of data storage. That job is performed by the RDBMS. The primary key uniquely identifies each row. One or more foreign keys can be established to provide connections to other tables. In this example, the Position Number attribute serves as a foreign key to a table storing position descriptions, where Position Number is the primary key. Foreign keys express association relationships. Cardinality is the ratio of rows for two tables. The number that comprises each half of the ratio is the table’s multiplicity. The cardinality of this one-to-many (1:m) association relationship says that a position number must be entered for each employee and that some position numbers may not be applicable to any employee.
Relational tables are not actually stored in the row-and-column form we typically use to visualize them, but everything you need to take from this book can be accommodated with the rows-and-columns metaphor. Oracle, SQL Server, Sybase, and Informix are commonly used RDBMS platforms. Products like Microsoft Access have much of the functionality of an RDBMS but are actually database management systems that employ files.

**Relationships in relational databases**

The big advance offered by the relational database is its ability to represent and manage relationships between tables. Where files normally use a record’s position in the file to uniquely identify each member, an RDBMS cannot impose any ordering on its member records. Thus, an RDBMS requires that at least one column be an instance identifier, called a primary key.

The relationship that relational databases are most concerned about is the association of one table to another. An association is established by placing the same column or a set of columns in both tables. This connection is called a foreign key. For example, a foreign key may link a central table storing general roadway information with other tables containing information about speed limit, traffic volume, maintenance jurisdiction, and pavement condition.

There is considerable variety in the nature of primary keys. The duty of a primary key is to uniquely identify each row in a table, which means that there can only be one row with a given primary key value. For this reason, many database designers argue against using a primary key that is entered by the user. This guidance also means the primary key cannot have any implicit meaning other than service as the row identifier. A primary key with intrinsic meaning is called an intelligent key.

Users like intelligent keys because they are easier to remember and they can serve double duty as an attribute. Database designers hate intelligent keys because they are prone to error in data entry and duplication within the database. You may want to use route number, such as SR 98, as the primary key. The problem is that you might accidentally type “RS 98,” or SR 98 might be rerouted, resulting in confusion as to the version a record references. Instead, database designers populate primary keys with integer sequencers supplied by the RDBMS and large, globally unique identifiers created through various mathematical processes. These values are guaranteed to be unique within the table.

All those other potential primary keys—the ones that mean something—are candidate primary keys and, thus, potential foreign keys. They could be primary keys, except for the chance that they might be duplicated within the table, which is the one thing that must never happen to a primary key. Coded values that are used as shorthand for a larger meaning, like a functional class code of 11 that means rural interstate highway, are often candidate primary keys that are chosen to serve as foreign keys. Some foreign keys may also be useful outside the database. These are called public keys, and they...
The two tables involved in an association relationship are called the origin and the destination. Both contain a field with the same data in the same form, although the number of instances with the same value may differ. The foreign key in the origin table is usually the primary key or a candidate primary key in the destination table. Association relationships are typically described as a ratio of the number of rows that can exist at each end of the relationship. Each number is called a multiplicity and the combination of the two multiplicities is called the relationship’s cardinality.

Multiplicity can be classified as one or many. Thus, cardinality can be the various combinations of these two values: one-to-one (1:1), one-to-many (1:m), and many-to-many (m:n). When the presence of rows at one end of the relationship is optional—in other words, the association doesn’t always happen—multiplicity can be zero, but that does not affect the cardinality. For example, if you designed a rail station database that contained a County table and a Station table, you must allow the number of Station table rows required for a given county to be zero, one, or more. It is, nevertheless, a one-to-many relationship because one county may have zero, one, or more rail stations. The upper bound in the multiplicity determines the cardinality.

In a one-to-one association, each row in one table may be related to one and only one row in the other table. This relationship is relatively rare because putting all the attributes in one table can often eliminate it. However, there are times when it is useful to split attributes of an entity into multiple tables. For instance, there may be a set of attributes that exist for only a small subset of entities or you may want to do different things with each subset of attributes.

The most common cardinality is one to many. In this case, one row in the origin table points to many rows in the destination table. The foreign key goes in the destination table and points to the origin table. For example, if you use coded values in your geodatabase, you will often provide a domain class that lists the range of valid values and ties each value to its
meaning. The table with value meanings is the origin and the table where those values are used is the destination. Many rows in the destination table can have the same value, all of which point to one row in the domain (origin) class.

The toughest cardinality to address in database design is many to many. This is because you cannot accommodate such an \( m:n \) cardinality by simply using a foreign key. A given row in either related table may need to point to an unknown number of rows in the other table, and a column in a relational database can only have one value.

Figure 2.4 The foreign key can go in either or both tables in a one-to-one relationship. Association is a connection that shows which tables participate in the relationship. Multiplicity expresses the cardinality of the relationship; i.e., the number of rows in each table that may participate in the relationship. The foreign key goes in the “many” end of a one-to-many relationship because that is the end with a single possible value. The foreign key in the destination table stores the primary key of the origin table.
To accommodate a many-to-many association, you have to turn the relationship into a table. Such an associative table will contain the primary keys of both related tables. The result is that each end of a many-to-many relationship can be listed in any number of rows in the associative table, thereby resolving the many-to-many relationship as a pair of one-to-many relationships. You can also store other information about the relationship.

Transportation databases are full of many-to-many relationships. For example, a work program project may affect several roads, and a given road may be affected by several projects over several years. Thus, the relationship between roads and projects is many-to-many. You will need a Road-Project table to store the relationship as a set of one-to-many relationships tying each project to one of the facilities it affected. Each row would have one road identifier and one project identifier. A given road or project identifier may occur in the associative table many times, but never more than once in the same combination.

Figure 2.5 Use an associative table to store the many possible relationships and give each one-to-many relationship its own foreign key. In a geodatabase, an associative table is called an attributed relationship class and can include user-defined columns.
Association relationships are usually obvious in their meaning, but naming relationships can help eliminate ambiguity. For example, you could say that an address includes a street name and that a street name is part of an address, or that an engine is part of a vehicle that may include many other components. We place relationship names next to the association connector and symbolize them with a caret that indicates the direction for which it applies.

Object-relational databases

The ArcGIS geodatabase is an object-relational design. Object-oriented software encapsulates data and the software that uses the data into an object class. The geodatabase consists of object classes that use a relational database approach to storing data. The kinds of object classes we will discuss are ArcObjects classes, which are the components of ArcGIS.

A class describes a set of data along with the functions that operate on that data. Class encapsulation is absolute. You cannot see inside the object class to view its data structure. You can only communicate with the object through the interfaces it supplies. An interface is a contract between the class and the outside world. Once an interface is declared for a class, it must always be supplied in every subsequent generation of that class and any classes that are based on that class.

Figure 2.7  Object relational  The ArcGIS geodatabase is object relational, which allows ArcGIS to evolve its internal class workings without affecting how you view the data. By convention, we treat geodatabase object classes as if they were composed of a single relational table, but they are much more. Appearance versus reality  With a geodatabase, ArcGIS only gives you the appearance of a relational database through its class interfaces and wizards. You are not actually seeing the internal data structure. A fundamental principle of object-oriented programming, called “encapsulation,” means that you can never see the internal structure.
Class interfaces provide a reliable way for software to be developed and used by different programmers. Each interface has a name. By convention, interfaces’ names start with a capital I with the subsequent letter also capitalized. For example, ITable, IClass, and ITableCapabilities are the three interfaces added by the Table class that is part of ArcGIS. Most interfaces build on others. The parental relationship would be expressed as child : parent, as in “IObjectClass : IClass,” which means that the IObjectClass interface offers additional functions for the IClass interface.

Object classes communicate with each other through messages conveyed between interfaces. You also communicate with an object class by sending, through an interface, arguments containing input data and work assignments to the class. The interface returns a result after the class’s software—called variously by the names of methods, behaviors, procedures, or operations—does its thing. ArcObjects programming involves the use of class interfaces. You cannot actually change the class itself because it is not possible to see inside the class due to encapsulation. As a result, ArcObjects documentation discusses only the interfaces and how to use them.

All the software you need to manage a geodatabase is not contained in one object or feature class but in dozens of ArcObjects classes that work together to provide the performance you need in a GIS platform. Drawing-layer symbology, attribute domains, relationships, and rules are also contained in separate ArcObjects classes and are just as much a part of the geodatabase as the object classes and feature classes you create. Data models do not generally include all the things you can specify for a geodatabase in ArcGIS. They are normally restricted to tables, feature classes, relationship classes, and domain classes. Things like how a feature class is displayed as a map layer are usually omitted.

To use any of the data models shown in this book, all you need to do is add fields to a class. No programming is required. You also do not need to know about or have experience with ArcObjects. In fact, this is the only place in this book where interfaces are discussed. The intent is to make you aware that a lot of things are going on when you use a geodatabase and to help you understand what you might see in ArcGIS documentation.

Unless you plan to do some ArcObjects computer programming, all you need to know to design a geodatabase is that you will define object and feature classes using a class template supplied by ArcGIS and add attributes, define rules, establish domains, and create relationships. You need not be concerned with how ArcGIS internally handles these parts of the geodatabase. However, because of the encapsulation of software and data within the geodatabase object-relational structure, you can only exchange data by sending the entire geodatabase to another user. You cannot just select one feature class and copy it. The behavior of that class depends on the contents of several other classes. A feature or object class functions only within the context of the geodatabase.

Just as software contained in an object class may be known by many names, attributes are also called by many names: columns, properties, and fields. There are more names for each discrete “thing” contained in a class: object, member, instance, row, and record. In general, ESRI restricts the use of object class to mean a type of table that stores nonspatial objects. This definition does not mean that the thing the object represents is nonspatial. A dam on a navigable waterway is certainly a spatial entity. If you choose to include it in a
geodatabase using an object class, you are only making the decision that this particular abstract representation does not include geometry.

In ArcGIS, an object that includes a shape attribute (geometry) is called a feature; i.e., it is a geometry object. In practical terms, a feature class adds the additional software and data structures needed to store and retrieve geometry to the software supplied by the object class. More specifically, a feature class is a table with a geometry column stored as binary long object (BLOB) pages in a relational database. A feature class contains geometric elements (simple features) or network elements (topological features) in a coverage, shapefile, or geodatabase structure. ArcGIS displays it to you as a table with a SHAPE column, but there is a lot more going on behind the scene.³

**Relationships in object-relational databases**

Our earlier discussion of relational databases introduced association relationships. Object-relational data models will explicitly display their multiplicity. Frequently encountered examples include “0..1” to mean zero or one may exist; “1” or no notation to show that one must exist (a default value); “*” to mean more than one must exist; “0..*” to represent zero, one, or more instances may exist; and “1..*” to mean at least one must exist, although more may be included. Association is represented in our models as a medium-width gray line and may carry role names.

The next most common and important relationship in an object-relational database is inheritance, which is a parent-child relationship. Inheritance is shown in a data model using a thin solid-black line with a generalization arrow pointing to the parent class. (You can think of the arrow as pointing in the direction to look for additional attributes.) The end symbol is called generalization because it shows the parent class to be a more general form of the child class, or, conversely, the child class is a specialization of the parent.

In addition to showing a parental relationship, inheritance simplifies logical data models by allowing you to omit repeating the parent class’s attributes in the child class. In our data models, the child class will include only attributes that have been redefined or added to the parent’s. Parent classes are often called stereotypes. Many stereotypes are abstract, which means that their purpose is solely to serve as a class template; objects of an abstract stereotype class cannot be created. An abstract class’s name will be shown in italics. A class that can produce objects conforming to the class specification is said to be instantiable.

The other primary relationships you will see in data models are dependency, composition, and aggregation. A dependency relationship, also called navigability, shows which classes depend on other classes for their existence. Another way to define this relationship is to say that one class instantiates the other, meaning a function of one class creates objects of the
Association  Association is a cardinality relationship between two classes that expresses the numeric ratio of how many of one class can exist relative to the other. Each end of the relationship includes notation for its multiplicity, except that convention omits the multiplicity of 1 as a default value that need not be written. The three basic cardinalities are one to one (1:1), one to many (1:m), and many to many (m:n). This example says that a culvert cannot exist in the absence of a related road, but that a road without culverts is possible.

Type inheritance  Perhaps you need to create a class with an extra attribute or two, or different implementation rules. The stereotype (Road) serves as a model for building two subclasses (Limited-access highway and Full-access highway) that share the attributes and methods included in the stereotype. In this example, the abstract Road class, which you will never instantiate, contains the attributes and methods that will be in both subclasses. The arrow-like endpoint symbol “points” to the superclass stereotype. Type inheritance makes logical models easier to read by reducing duplication.

Instantiation  Some classes can create instances of other classes through instantiation. For example, a Road object might be able to create a Right-of-way object. A data model that consists only of standard feature, table, and relationship classes will not include instantiation, as ArcGIS handles those duties.

Aggregation  Aggregation is when an instance of one class (the whole) represents a collection of instances in another class (the part). A DOT work program, for example, could be viewed as a collection of construction projects. The project exists independently of the work program.

Composition  In contrast to aggregation, where both classes involved in the relationship can continue to exist in the absence of the other, composition means that the “whole” class controls the existence of the “part” class. Here, if you delete a Road class instance, all the related Pavement segment instances will also be deleted.

Figure 2.8  Relationship types  Data models at all levels of abstraction include relationships. This minitutorial explains the five primary types that you see in ArcGIS: association, type inheritance, instantiation, aggregation, and composition. All relationships are illustrated using a line and endpoint symbols.
other class. Dependency is shown using a dashed black line with an arrow pointing to the dependent class. None of the data models presented in this book will include dependency relationships, although they do frequently appear in ArcGIS documentation of the geodatabase and you may need them in your own data models.

Composition and aggregation are similar to each other with one important difference. A composition relationship is created when one class is composed of one or more instances of other classes. For example, a building may be seen as being composed of at least three walls, one floor, and one roof. Remove the building and its components cease to exist, at least as far as the database is concerned. Thus, a composition relationship tells you that when the sum of the parts is deleted from the database, you will also need to delete the objects of which it is composed. A thin black line with a solid-black diamond at the end adjacent to the composite class represents this kind of relationship.

Aggregation is not so particular. An aggregation relationship specifies that a class is a collection of other classes. For example, a baseball team may be a collection of players. If you express this relationship through aggregation, then you are saying the players will continue to exist in the database even when they do not belong to a team. If you instead use composition, then the players must be deleted from the database when the team is dissolved. Aggregation symbology is similar to that for composition, except that the diamond is outlined in black and white filled.

The data-modeling process

You need a good data model to produce a good geodatabase design. Developing a geodatabase design is a six-step process that follows the flow of the agile methods discussed in chapter 1:

Step 1—Define user requirements. First, you need to know the purpose of the data, the application requirements to be supported. Many users attempt to develop a complete set of requirements as the first step, but that cannot be done. Even for a small project, the agile method instead encourages you to create a good first effort that has the primary objective of identifying the major components. For an enterprise geodatabase that will support a wide variety of existing and yet-to-be-created applications, seeking a complete set of requirements as the first step is an impossible goal. No, your task here is to identify the general requirements.

Step 2—Develop conceptual data model. Once you have specified the general requirements for the final product, you will need to identify the basic elements of a geodatabase that meets the requirements. Such elements consist of entities and their relationships. An entity may eventually be reflected in a class, but at this point in the process, you cannot establish a one-to-one equivalency between entities and classes.
Chapter 2: Data modeling

Step 3—Develop a logical data model. Once the general structure of the database—the skeleton—is established, the next step is to add some meat to the bones by specifying attributes for the geodatabase. Entities may change at this point, as attributes are assigned and new relationships discovered. The logical model is independent of the planned implementation platform.

Step 4—Develop a physical data model. Here is where entities become classes and the implementation platform makes a difference. Your RDBMS, network structure, and organizational behaviors will influence the way you translate the logical design into a physical implementation. The added benefits of the geodatabase and ArcGIS will become apparent at this stage. The geodatabase can perform many functions that would normally have to be handled by user-developed software. For many geodatabase projects, the first task will be to split entities into tables and feature classes. You will also need to decide which fields can be supported by domain classes and which relationships need to be instantiated as a relationship class, not just an implicit relationship established by foreign keys you use when you desire. You need to be alert to the difference between layers on a map and components of the geodatabase. The next task is to specify the details of each class and create the domains, rules, and other elements of a geodatabase. The physical data model specifies the data type, default value, domain, and other characteristics of each attribute. The logical data model tells you about the classes and their attributes, although you may not implement the whole model at one time, and tests may motivate changes to get the desired performance.

Step 5—Test the data model. Next, you can load the physical data model into ArcGIS and generate a prototype database for testing. Many central elements of a transport geodatabase can be implemented in more than one way. Testing the prototype before you put it into production is a good way to evaluate the efficiency of the implementation choices you made. Testing should include typical editing operations and involve a sample dataset equivalent in size to the one you will use. If the design does not pass this test, it may be necessary to go back to step 3 or 4 to make other choices, but it is much better to find out now than after you put it into production use.

Step 6—Production implementation. Now you can reap the rewards of your work. Load the geodatabase and create the default version. It is time to put everyone else to work.

These steps are generally sequential but you may move backward whenever necessary to redesign a portion of the geodatabase. You may also choose to prototype parts of the design at points well before step 5 so as to test key components. What works great for one agency may be a bad idea at another because of an organizational difference or the combination of applications to be supported. It is much cheaper to debug a paper design than an implemented geodatabase. Modeling will not eliminate all chance of error, but it certainly improves the odds of success. The balance of this chapter will explore the differences between conceptual, logical, and physical data models at one time, and tests may motivate changes to get the desired performance.
Conceptual data models

Data models typically go through three phases of increasing specificity, starting with conceptual modeling. This phase is primarily concerned with identifying the entities about which you will need to retain data and the relationships between those entities, including some that will also need descriptive data. Conceptual modeling considers the application the data is to support and defines database terms. For example, if you are developing a geodatabase for a transportation agency, you will need unambiguous definitions for terms that will appear in the model. This set of definitions is called an ontology.

Here is where you should expect your first philosophical debate. You’ll find such common entities as Road, Railroad Track, Bridge, and Airport will often have very different meanings throughout the agency. What do we mean by Route? Is it the continuous piece of pavement that winds through many states, each one assigning its own name to it? Or does the name itself define the extent of a route? If the latter, what happens when the name is changed or the route takes a different path due to construction? How about if the road is realigned in some way so that the length changes? Does Road include Right of Way as an element, or is Road an element of Right of Way? Is Airport a piece of land, a terminal, a collection of runways, or an airspace? Is Railroad Track one set of rails, with a section of double-tracked mainline being two Railroad Track members, or is Railroad Track like Road, where each track is equivalent to a lane of traffic and the number of tracks is an attribute? Is a Bridge across an interstate highway part of the interstate or the road that crosses the Interstate? Answering such questions is intense, emotional, and necessary. You will quickly discover that the most important relationships are those in the room, not those shown in the data model.
A conceptual data model shows entities and their relationships. It does not include attributes. A conceptual data model expresses central concepts, illustrates data structures, and describes components of the ArcGIS object model. You will use conceptual data models to translate user requirements into data structures. Creating the data model usually begins the process of developing the application ontology, which includes formal definitions for all the entities, attributes, and operations that will be part of the final design.

![Diagram of conceptual data models]

**Figure 2.9 Conceptual data models** The intent of a conceptual data model is to express the entities and relationships in a highly abstract manner. Attributes and methods are not included in a conceptual model, so the complex notation of UML is not required. Indeed, it may serve to obscure the model's meaning.

Figure 2.9 illustrates the simple 2D and 3D boxes used for conceptual data models. This is the same graphical standard used for many ArcObjects diagrams contained in ESRI documentation. For our purposes, conceptual data models consist of entities, not classes, and no one-for-one equivalency should be assumed; however, ArcObjects models presented with the same symbology do have equivalencies between entities and classes. In both cases, entities will be shown as one of three types. An abstract entity will be shown with a 2D rectangle and the name in italics. Instances of an abstract entity will not be implemented. Abstract entities form stereotypes for other entities that can be implemented.

Entities that are not abstract will be shown using a 3D cube, with a slight difference in face color between instantiable and creatable entities. This distinction really only applies to
conceptual ArcObjects models representing a class structure. All non-abstract ArcObjects classes are instantiable in that members of each class (objects) can be generated. An instantiable class, as that term is used in this context, is one that is creatable only by other ArcObjects classes. Members of a creatable class can be instantiated by the user directly through ArcGIS. True conceptual data models will include only creatable entities, because users cannot generate instantiable entities directly.

The figure does not include notes and callouts, two of the most useful parts of the conceptual data model. ESRI’s standard notation shows a finished product. What you are creating is a work in progress. You need to add notes that explain what the model says and callouts to describe specific entities and relationships. Business rules and definitions are not generally part of published conceptual models, but you will need them. The only important consideration is that the team members developing the model understand the model. Do not try to adhere to a particular external documentation standard for everything. This is not the time to try to teach everyone about the details of data modeling. There is no extra credit for pretty pictures. Use what works best for you. This book shows you the part that probably should be fairly uniform across teams and will be consistently used in this book.

Logical data models

Logical data models presented in this book use a simplified version of the iconic notation typical of Unified Modeling Language (UML) steady-state diagrams. These diagrams present classes and their relationships. You have already seen the UML relationship notation. You also know a class is the encapsulation of software and the data it needs. We only want to model data. ArcGIS and the geodatabase take care of the software part.

Figure 2.10  Object classes  This is the normal form of UML steady-state data models. Visibility defines the degree to which class attributes and methods are accessible by other classes. This standard symbology is modified to create the graphical standards for logical data models. The two primary changes were omitting methods, which the ArcObjects we might include in the database design already provide, and assuming that all attributes (properties) are open for anyone to see (public). All attributes added to geodatabase tables and feature classes are also public.
Our data model graphical standards represent a compromise between graphical consistency with UML and other ArcGIS documentation standards. UML is actually used by computer programmers to design their software. The notation is adapted here for use as a data modeling language.

Although it is common to do so, it is really a bit of a stretch and mismatch to use UML steady-state diagrams as a data model. UML is really for application design. Since the data and software that work on the data are tightly bound through encapsulation, steady-state diagrams do show the data, but with a limited view.

Normally, a class symbol is a rectangular box subdivided into three parts. The upper part of the class box holds the name of the class. The center part of the box holds the various class properties (data). The bottom part holds class methods (software). However, the bottom part is not required since ArcObjects classes already include the operations needed to implement a geodatabase. So, logical data models for geodatabase designs use a rectangular, two-part box that omits the bottom methods section. All class properties that you add will be public, so the visibility indicator is not required. Class properties can be referred to as attributes, fields, or columns, and such terms as class, table, and feature can be used to refer to the entities in logical data models.

OK, here is the truth: the other sidebar is lying. You do not actually add properties, in the UML sense of the word, to any ArcObjects class when you create a geodatabase. What you are really doing is using properties and methods that are already in the class. The ArcGIS user interface allows you to access those properties and methods by using wizards and other tools to customize classes so they serve your purposes, but you are not really changing those classes. UML steady-state diagrams are for creating software. You are not creating software; you are designing a geodatabase that is constructed of classes provided by ESRI and, perhaps, one or more of its business partners.

Think of it like using spreadsheet software. When you get started, all you see is a bunch of little boxes into which you can type numbers, text, and formulas. Anything that happens with the contents of a box is already in the spreadsheet software. You are not creating the spreadsheet software. It is the same with the geodatabase you are designing. The geodatabase is already in ArcGIS. You are just telling it which of its capabilities to use and what the inputs and outputs should be.

So, UML is really a poor way to create data models, but it is the one we have. If we were starting from scratch to create a new language, we could definitely think of something better than English, but then we would not have anyone to talk to. It’s the same way with using UML to create data models. It is a language many people already understand.

In all cases, the name of a class that can be created will be shown in normal type, while the name of a class that is abstract in nature (a stereotype that cannot be created but serves as a template for defining a creatable class) will be shown in italics. Attribute names will be
stated using a concatenated mnemonic name in a Roman sans serif font with an initial upper-case letter and intermediate capitals to assist in understanding, for example, FirstName.

Besides the entities, a logical data model includes relationships. Relationships are shown as a line with end symbology. For example, the UML diagram in figure 2.10 shows an inheritance relationship, which means that the Employee class is based on the Person class. By convention, UML only shows the new properties and methods added in the Employee class, with all the Person class properties and methods included by the inheritance relationship. Person is thus the parent stereotype or supertype of the Employee class.

Sometimes the supertype class is not shown. By UML convention, the name of the superclass from which the class inherits its base attributes in the class name space can be included in the entity name space. Some classes may include subtypes listed below the normal class specification. All subtypes have the same attributes.

Figure 2.11  **Stereotypes**  Inheritance is normally shown through an explicit relationship, but it can also be indicated by placing the name of the superclass in the class name space within double carets. This convention is normally used when the superclass and its inheritance relationship are not shown.

Figure 2.12  **Logical data model**  A logical data model fleshes out the entities of the conceptual model by adding attributes and resolving many-to-many relationships. Since an ArcGIS geodatabase consists only of predefined classes with mandatory and user-defined attributes, methods will not be included. The result is an abbreviated form of the traditional UML notation and the usual reference to the class properties as attributes. Each ArcGIS class will determine the manner in which attributes are converted to properties.
Relationship notation continues unchanged from that of conceptual data models, but there are differences. One change you should notice is that there are no many-to-many relationships in a logical data model. They have to be resolved during the transition from conceptual to logical form. Otherwise, a logical data model will look much like a conceptual data model with attributes added. A logical data model may also include an enumeration of values that help express the domain of one or more attributes. An enumeration is an example list of values for a domain. The complete domain does not have to be specified until you create the physical data model. The enumeration may become a domain class in the physical data model.

**Physical data models**

The most complete version of your geodatabase design is the physical data model, which includes many of the bells and whistles a geodatabase can supply. Classes are more precisely specified, as are their attributes. As with the transition from conceptual model to logical model, changes in the design may occur as you construct or test the physical data model.

The core of any physical geodatabase model will be object and feature classes. Relationships may be implied by foreign keys or explicitly included as relationship classes. An implicit relationship is called a join relationship and represents association. Explicit relationships may include attributes or merely enforce cardinality rules.

Domain classes may be added to control data entry by limiting the available choices to a defined set. An enumeration of representative values included in a logical data model must be converted to a fully defined list of values for the physical data model if it is to be reflected in a domain class.

The geodatabase has rich capabilities that ease the transitional leap from a conceptual to a physical data model. In the past, you would have been required to break down entities in the conceptual model into component parts when you made the transition. The implementation environments for which the physical data model is designed required you to provide the behaviors and data structures necessary to express the full range of characteristics and actions embodied in each entity. For example, with a relational database implementation, you have to create lookup tables for domain control and manage association relationships with software you write. In contrast, the underlying geodatabase data model allows you to implement such behaviors by simply declaring the domain of valid values and stating the rules you want to enforce, all without writing any code at all. At the end of the day, the geodatabase classes you define in the physical data model for a geodatabase implementation look very much like the entities in the original conceptual data model.
The next chapter describes the geodatabase and how it works. It also presents some basic techniques of geodatabase design.

The physical data model exists to embrace the implementation environment and mold it to the form required by the logical data model. This example is for an ArcSDE geodatabase that includes road centerlines in a polyline feature class, plus states and the routes they contain in two tables. An attributed relationship class handles the many-to-many relationship between State and Route: a state can contain many routes and given route can traverse many states. Four coded-value domains have been included to manage data inputs.

All the data models included in this book were created using Microsoft Visio, which is also supported by ArcGIS for loading database designs into ArcCatalog so as to create classes automatically. Instructions for how to do this are contained in the online ESRI Support Center. There are many tools you can use to create data models. If you have a copy of Visio, you will notice that it contains many templates for software and database design following a wide variety of published “standards.” Use the ones that work for you.
Notes
1 A computer scientist will tell you that these structures are called relations, not tables, and they consist of tuples (rows) and vectors (columns). These terms were chosen to avoid the impression that the data is physically stored as rows and columns in a separate piece of the database. In most RDBMS implementations, all the records in all the tables are stored in one big file. From a database design perspective, it is much better to work with tables containing rows and columns than the more ephemeral concepts of tuples and vectors contained in a table space.
