



DATA MODELING

SECOND, REVISED EDITION

BY MICHAEL D. WALLS



The Urban and Regional Information Systems Association
1460 Renaissance Drive, Suite 305
Park Ridge, IL 60068

Copyright ©2007 by the Urban and Regional Information Systems Association (URISA), 1460 Renaissance Drive, Suite 305, Park Ridge, IL 60068, (847) 824-6300, www.urisa.org.

All rights reserved including the rights of reproduction and use in any form or by any means, including the making of copies by any photo process or by any electronic or mechanical device (printed, written, or oral), or recording for sound or visual reproduction, or for use in any knowledge or retrieval system or device, unless permission in writing is obtained from the copyright proprietor.

Printed in the United States

ISBN #: 0-916848-14-0

Table of Contents

ABOUT THE AUTHOR	5
INTRODUCTION	7
SECTION 1 — Introduction	9
SECTION 2 — What is Data Modeling.....	11
The Context in Which Data Modeling Occurs	13
Formal Definitions	15
Database Design Methodologies	15
The Data Modeler’s Skill Set	16
Definitions of Essential Technical Terms	17
What Is an Entity?	17
What Is an Attribute?	18
Primary Keys.....	18
Foreign Keys.....	18
Nonkey Attributes	19
What Is a Domain?.....	19
What Is a Relationship?.....	20
Superentity/Subentity Relationships	21
Putting the Pieces Together—How a Relational Database Works	22
“Store Data Once, Use Many Ways”	22
The Relational Join	22
The Geospatial Join	22
Why We Bother with Data Modeling.....	23
Problems of Appropriateness in Design	23
Problems of Definition.....	23
Problems during Implementation	23
Problems of Correctness	24
SECTION 3 — Performing Data Modeling	25
A Process for Data Modeling	25
Conceptual Design—Defining the Model’s Scope.....	25
Logical Design.....	27
Fully Attributed Model	29
Physical Design	30
Physical Implementation	35
Normalization and Denormalization	35
The Normalization Process	36
SECTION 4 — What Do You Do With a Data Model?.....	41
SECTION 5 — What’s Different About Spatial Data?	43
How GIS Is Different from Other Information Technologies	43
How GIS Integrates with Other Data Management Technologies.....	46

Adjustments to Database Design Process.....	47
In Conclusion	48
SECTION 6 — Whats Different About Object Modeling?	49
Overview of Object Orientation	49
Database Design for Object Orientation.....	50
Impacts of Object Orientation on the Approach in This Book.....	52
Appendix— Resources for the Data Modeler	55



ABOUT THE AUTHOR

MICHAEL D. WALLS SENIOR DATA ARCHITECT ATOS ORIGIN

Mr. Walls currently works as a data architect for a business intelligence and data warehousing consultancy, for Fortune 500 clients. He had previously worked over nine years as an Executive Consultant for PlanGraphics, Inc., a GIS consultancy.

Mr. Walls has also had over 20 years experience in local government, with agencies ranging from an Indian tribe to universities to a regional river authority. Most of that time was in city and county planning departments. He has degrees in geography, anthropology, public administration, and computer science. He also has received professional certification from the American Institute of Certified Planners and the Project Management Institute.

Mr. Walls began his career as a grantsman, project planner, and evaluator. While working as a public

policy analyst, he became involved in computer programming. Over time he moved from using computers to do his own planning tasks, to helping other planners, to developing applications for general use. He has managed an applications development team for a city planning department, and established and managed a GIS Department for a state river authority.

As a consultant, Mr. Walls specializes in data architecture and software engineering. He has been involved in a wide variety of IT studies, including data warehousing, data provisioning and ETL, enterprise architecture planning, data architecture and database design studies, and reporting and other applications development projects.



DATA MODELING

INTRODUCTION TO REVISED EDITION

I wrote the first edition of this *Quick Start* monograph in 1999 with the intent of describing to fellow practitioners what I'd learned about data modeling and related design activities. The core message was that data modeling is interesting and an essential skill for information technology workers, and it is really not that difficult to achieve a workmanlike data resource design. (Indeed, I have been known to claim that data modeling is about the most fun you can have with a computer, but friends usually convince me that folks won't believe me if I say it in public. I don't know why not, but . . .)

When I was invited to update this material, I thought a great deal about what I might want to change in light of an additional seven years of increasingly intensive specialization in data resource management on my part, and in light of significant changes in the database management system (DBMS) technologies available to us. I immediately decided that this core message is still correct—if anything, the changes I've seen only strengthen the interest and the importance of data modeling. The level of difficulty has increased for me, to some extent because of the need for an old dog to learn new, object-oriented tricks. If you are new to data modeling, you may find the current technologies actually make things easier for you.

So, what is new or different in this revised edition? First, I've taken the opportunity to make minor improvements in wording and to correct the occasional typo throughout the body of the material. Then, the “new stuff” consists of:

- This introductory material
- A new Section 5 explicitly discussing geospatial data. Although I have specialized in geographic information systems (GIS) through much of my career, I have always downplayed the differences between GIS and other types of data resource management. URISA members work in a broader data resource management arena than just GIS, and I wanted to support this larger community. However, the majority of feedback I've received from readers of the first edition has focused on GIS applications. Rather than rework the more general material, I've added this section to try to answer the question, What's so spatial about geospatial data?
- A new Section 6 explicitly discussing object-oriented (OO) data resource management. This topic has greatly increased in importance since the first edition because of three

technology trends. First, the widespread adoption of the Unified Modeling Language (UML) in software engineering has focused attention on OO practice, even while obscuring the often contentious relationship between OO approaches and data resource management. Another trend has been the increasing mainstreaming of OO approaches to DBMSs through inclusion of object extensions to relational database management system (RDBMS) products such as

Oracle. (If all these acronyms and terms are confusing, don't worry, the early sections of the document introduce these as needed.) A third trend has been the increasing move towards OO approaches by GIS vendors. In this section, I try to indicate how I think object orientation fits within the overall data modeling process, and where OO requires different approaches (and where it does not).

- An updated References section



SECTION 1

INTRODUCTION

Early in my career I was very frustrated at not being able to find a clear and concise guide to data modeling and database design. Academic studies either concentrated on using existing databases or on details I would only need to know if I developed my own database management system from scratch. Available textbooks offered a chapter or two on logical design, but generally brushed over the topic as something reserved for specialists.

Over the years I've become one of those specialists, through a combination of education, self-study, and on-the-job training. The more I work with and study data modeling, the more I am convinced that much of the mystery around the topic is unnecessary. This guide is my attempt to present what I've learned about this discipline to a nonspecialist audience, whether information technology (IT) specialists who are not data resource managers or the people who will ultimately rely on the data repository to do their work. A primary focus is on “how-to” instruction that takes the reader through the steps of creating a data model.

This document has the following objectives:

- To concisely present enough technical details about the data modeling and (to a lesser extent) database design processes to allow the untrained reader to interact with computer professionals (in print or in person) without being totally lost.
- To present an applied approach to data modeling that allows a novice to avoid most pitfalls in designing and using complex databases.
- To describe the position of data modeling within an overall system planning and development progression.
- To explain the purposes of data modeling—a means to build consensus, a foundation for detailed database design, and a support for applications design.



SECTION 2

WHAT IS DATA MODELING

Before we get into instructions for performing data modeling, we need to discuss what data modeling is. We also need to differentiate data modeling from process modeling or other types of modeling, and to deal with the distinction between data modeling and database design.

We will start somewhat informally and subsequently provide a more formal definition. I think of **data modeling** as defining an abstract, summary representation of the real world that will form the basis of a physical database design. In other words, the data model is the bridge between the real world as seen by the enterprise and the information-processing systems that will support the enterprise in doing its work. **Database design** is the process of specifying a set of data structures that can store and manage attribute values about one or more instances of real-world things, in compliance with the data model. This physical database design will typically be implemented on one or more computers, using some database management system software, but may rely only on paper forms.

(I suppose that given the definitions I'm using, *physical database design* is redundant. The term is widely used, however, within a conceptual framework that also includes talk of logical database design. I have

come to use terms that emphasize the abstraction of the logical model from the real world versus the design of the physical implementation.)

(At this point, let me also address a sometimes confusing side issue. Throughout this document I will use the terms “enterprise” or “business”. I have had resistance from academics and nongovernment organizations (NGOs) to the use of these terms. This terminology does not mean that data modeling and related techniques are restricted to use in business! For example, any organization may be described in terms of “business rules.” An *enterprise* may be a retail merchant chain, a governmental agency, a club or other social organization, or even an individual. As an interesting note in the sociology of organizations, I've had similar complaints to the alternative terminology, *agency*, from business clients!)

Data modeling is an essential aspect of most information technology projects, for it provides a means of determining just what data is required for a successful project. It is one stage of the overall IT project planning and design life cycle, but supports many of the other stages. The ultimate value of data modeling is the assurance that we are working with a complete and accurate picture of the data required by the enterprise to perform the specified set of activities.

Figure 1

ENTERPRISE ARCHITECTURE - A FRAMEWORK TM

	DATA	INFO	FUNCTION	HOW	NETWORK	WHERE	PEOPLE	WHO	TIME	WHEN	MOTIVATION	WHY
SCOPE (CONTEXTUAL)	Is it/Things important to the Business? 	Is it/Functions the Business Performs? 	Is it/Locations in which the Business Operates? 	Is it/Organizations important to the Business? 	Is it/Process Significant to the Business? 	Is it/Issues Critical to the Business? 	SCOPE (CONTEXTUAL)					
Planner	Priority = Place of Business; Things 	Function = Choice of Business; Processes 	Mode = Major Business Locations 	People = Major Organizations 	Time = Major Business Hours 	End/Issue = Major Business Critical Issues; Risks 	Planner					
ENTERPRISE MODEL (CONCEPTUAL)	e.g. Semantic Model 	e.g. Business Processes; Model 	e.g. Logical Network 	e.g. World Business Model 	e.g. Major Schedule 	e.g. Business Plan 	ENTERPRISE MODEL (CONCEPTUAL)					
Owner	End = Business; Entity; Risk = Business Relationship e.g. Logical Model 	Proc = Business; Process; Prod = Business; Production e.g. "Requirements Architecture" 	Mode = Business; Location End = Business; Location e.g. "Information System Architecture" 	People = Organizations; Unit Who = Who? Product e.g. Human Resource Architecture 	Time = Business; Event Cycle = Business; Cycle e.g. Performance? Structure 	End = Business; Objective Risk = Business; Risk Model e.g. Business Risk Model 	Owner					
SYS TEAM MODEL (LOGICAL)							SYS TEAM MODEL (LOGICAL)					
Designer	End = Data Entity; Risk = Data Relationship e.g. Physical Data Model 	Proc = Applications; Functions Who = What? How? e.g. "System Design" 	Mode = IT Function; Processes; Services; Applications End = Data; Services; Applications e.g. "System Architecture" 	People = Roles Who = Who? Product e.g. Process Architecture 	Time = Event Cycle = Component/Cycle e.g. Control Structure 	End = Structural/Service Risk = Service/Service e.g. Risk Design 	Designer					
TECHNOLOGY MODEL (PHYSICAL)							TECHNOLOGY CONSTRAINED MODEL (PHYSICAL)					
Builder	End = Equipment/Job; Risk = Equipment/Job e.g. Data Definition 	Proc = Computer Function; Prod = Software/Device Function e.g. "Program" 	Mode = IT Service/System Software End = Data; Services; Applications e.g. "Network Architecture" 	People = Users Who = Who? Product e.g. Service Architecture 	Time = Event Cycle = Component/Cycle e.g. Time/Duration 	End = Condition Risk = Action e.g. Risk Specification 	Builder					
DETAILED REPRESENTATIONS (OUT OF CONTEXT)							DETAILED REPRESENTATIONS (OUT OF CONTEXT)					
Sub-Contractor	Proc = Business; Risk = Business 	Proc = Language; Unit Who = Component/Block 	Mode = Software; End = Product 	People = Agency Who = Who? 	Time = Activity Cycle = Activity 	End = Sub-Condition Risk = Risk 	Sub-Contractor					
FUNCTIONING ENTERPRISE	e.g. DATA	e.g. FUNCTION	e.g. NETWORK	e.g. ORGANIZATION	e.g. SCHEDULE	e.g. STRATEGY	FUNCTIONING ENTERPRISE					

The Context in Which Data Modeling Occurs

John Zachman has provided a conceptual framework within which we can place the activities we call *data modeling*. His most recent version of what has come to be called the Zachman framework (Zachman 1987) is shown in Figure 1. Data modeling focuses on the aspects of the enterprise described in cells comprising the Semantic Model and Logical Model within the Data column of the framework. One of the biggest benefits of the Zachman framework for data modelers is the justification it provides for excluding work flows, architecture decisions, and other aspects of the enterprise architecture from the data modeling task. (Other documents in the *Quick Study* series focus on adjacent cells.)

The Zachman framework, in common with many data modeling methodologies, stresses an enterprise approach to the problem. Other approaches, especially those coming from a user perspective or from an applications development background, focus on a particular problem to solve. The data modeling literature is rife with discussions of data-centered versus function-centered design—virtually all of which emphasize data-centered approaches. (In contrast, the object-oriented literature concentrates on the functionality of the object, sometimes at the expense of any data content.) Although I agree in principle, I think one can have a too simplistic data-centric bias.

My answer to which approach is best is that “it depends.” In principle, the enterprise data model is best, for it better ensures that the investment in capturing and managing data will yield the greatest benefits, and because it cuts down on the risk of inconsistent answers within the enterprise. However, when on a tight deadline with high-priority problems, you cannot always wait for a perfect solution and must take an application focus to solve the specific and immediate problem. I would urge you to constantly be looking for ways to make your work immediately,

concretely beneficial to the enterprise. In the GIS arena, we’ve learned the hard way that projects that take more than about 18 months to create some obvious positive impact are probably doomed.

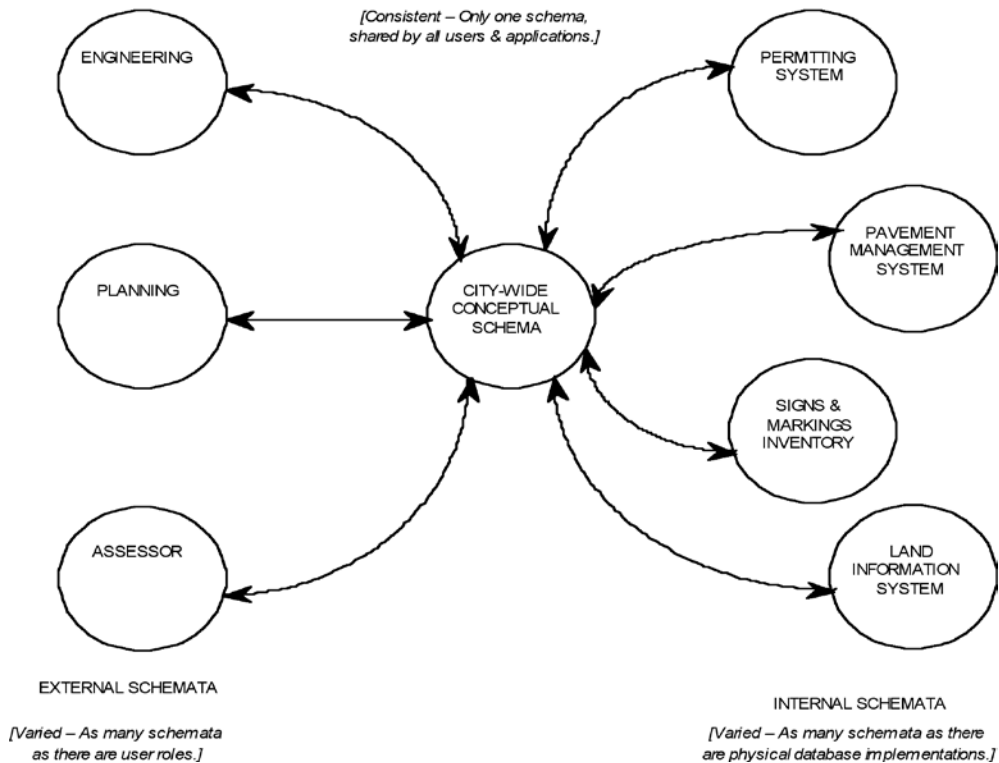
I find a function-centered approach is usually best for solving a particular business problem, while a data-centered approach is best for facilitating an enterprise-level project. However, a data-centered project should be implemented with appreciation of the value and reusability of the data required to support the function. If the function is well understood, then the data requirements to support that function will by definition be valid from the enterprise perspective. The importance of the data orientation lies in: (1) planning for maximum data reusability and (2) choosing among equally satisfactory (from the function perspective) alternative data structures the one that will be most generally useful to the enterprise.

In theory, a set of well-designed data-intensive applications that were tightly integrated with one another would yield something approximating the enterprise data model—through a kind of “bottom-up” specification process. (In practice, though, you would have to do a lot of scrap and rework to get things to integrate.) To see why this is possible, we can use the classic three-schema conception of database design.

In the 1970s, there were several competing architectures for database management. It was difficult to share understanding between teams within the enterprise unless they all used exactly the same architecture. The American National Standards Institute (ANSI) in 1975 sponsored a committee to help organize the situation. The results were reported in (ANSI 1975), and one key finding is summarized in Figure 2. Basically, the committee brought order to the confusion by recognizing that people had been talking about different aspects of the problem as though they were the same. The model they came up with recognized:

Figure 2

THE THREE-SCHEMA DATABASE ARCHITECTURE



- **External schemata.** Every user group has different understandings of the operating environment and, therefore, of the data required to support their work. In computer terms, these understandings are embodied in the data screens, database views, and the computer programs that make up an application. There can be many of these perspectives within the enterprise, each represented in a schema.
- **Internal schemata.** Each of the many data resources used within the enterprise has a physical structure, hopefully one that is well designed and well documented. In computer terms, these will be database tables, spreadsheets, or GIS layers. This structure is represented in an internal schema.
- **Conceptual schema.** The business logic will be more or less consistent across the enterprise, although varying with the integration

and relatedness of business functions. (This is why governmental agencies have greater difficulty than most businesses in enterprise data modeling—there is a limited relationship between paving streets and operating health clinics, for example, while a restaurant chain has almost every function focused on providing food and related services to customers.) There will be only one conceptual schema for an enterprise, although getting agreement on its content can be a major challenge.

A many-to-many relationship (a concept we will formally define a little later) exists between external schemata and internal schemata. That is, several external views can be implemented simultaneously in both a GIS layer and a spreadsheet. However, there should be only one conceptual schema within the enterprise. It is the process of identifying the contents and specifying

this logical structure that we refer to as *data modeling*. In contrast, *database design* is the specification of a single internal schema. *Software design* is usually the process that defines a set of external schemata, though the inclusion of database views within the external category encourages database design to consider some aspects of the external schema.

Formal Definitions

Data modeling. The process of generating an internally consistent conceptual schema of the enterprise or problem environment, which is a valid representation of both the set of real-world things deemed relevant to the enterprise and of the business rules that control the enterprise's interactions with these things.

Database design. The process of specifying the physical database that will provide one mechanism for efficiently and effectively storing and utilizing facts about real-world occurrences of those things of interest to the enterprise. Database design presumes the existence of a data model to use as its input specification and of a data-management tool (typically a relational database management system or GIS) that provides technical guidance for and constraints on the design. The tests of goodness of a database design are the quality of representation of the data model and its implementability within the target technology.

Data repository. The physical implementation of a set of data storage structures into which data can be stored. This may be a GIS coverage, layer, or library. It can be an RDBMS instance or an operating system file. (The term is specifically chosen to be implementation-neutral—no assumptions about the technology to be used are being made. In fact, a data repository could be a paper form filled out and stored in a file cabinet.)

Data resource management. The processes used to design, implement, and maintain data repositories (as defined previously) to maximize their effectiveness in supporting their intended purposes and the

goals of the enterprise(s) that implement them. These processes are a mixture of technical methodologies and management practices.

Database Design Methodologies

When we set up a data repository we are representing only a selected subset of reality. Many possible solutions normally exist for a real-world problem. The only true measure of quality for a data modeling exercise is whether or not the resulting data repository meets the needs of its users. It will do so if two criteria are met:

- Does the model contain useful representations of all the important aspects of reality that a user must know to solve his or her business problem?
- Does the database avoid including a lot of extra information that slows down getting the answer?

The data modeling discipline, like other aspects of software engineering, has generated several formal methodologies. In my opinion, almost all methodologies have the same general sequence of events, but different practitioners have preferences in tools and terminology used and for the breakpoints between the recognized stages. The reader is encouraged to explore various methodologies, to find one that suits his or her style and situation. Some important methodologies include:

- **Entity-Relationship Modeling.** This was perhaps the earliest formal data modeling methodology. Created by Dr. Peter Chen, this methodology has been broadly influential through the near universal adoption of its primary technical aspect, the notation of Entity-Relationship Diagrams. Over the years, various extensions to the basic methodology have been proposed, including Semantic Data Modeling. (See Chen 1977; Byte 1989 in the Resources section.)

- **Structured Methods.** This refers to the data-oriented components of the suite of structured methods for applications development that arose as programmers became dissatisfied with hit-or-miss approaches. These encouraged separation of logical and physical design issues, and a systematic top-down approach to developing what today would be called application-oriented systems. (See Atre 1980; Orr 1977; Byte 1989 in the Resources section.)
- **Information Engineering.** IE was jointly developed by James Martin and Clive Finkelstein, who subsequently diverged in their approaches. IE approaches (to perhaps oversimplify) use the structured-methods approach applied to an enterprise perspective rather than an application-specific perspective. Specific applications are approached only after a sufficient understanding of the enterprise is developed. (See Finkelstein 1992; Martin 1990 in the Resources section.)
- **Object Methods.** The concept of objects first took hold in the programming community, but soon moved into database management system concepts. Objects provide yet another type of internal schema for representing a conceptual schema, and it can therefore be argued that any data modeling methodology can be used to define a data object (although additional work may be required to understand its methods). However, several general-purpose object-oriented modeling techniques have been devised, three of the most prominent of which have been merged into the quasi-methodology, the Unified Modeling Language. As the name implies, UML is primarily a notation and requires additional techniques to decide what content to represent. (See Eriksson and Penker 1998 in the Resources section.) This will be discussed in more detail in Section 6.
- **IDEF1X.** IDEF1X—pronounced “Eye-Def

One X”—originated with a U.S. Air Force attempt in the mid-1970s to perform large-scale database design in the context of an integrated computer-aided manufacturing (ICAM) program. The acronym stands for “ICAM Definition, Part 1, Extended.” It is part of a family of techniques for business modeling and database specification, and has been adopted as a Federal Information Processing Standard (FIPS). (See NIST 1993; Bruce 1992 in the Resources section.)

The approach to data modeling I have devised over the years is primarily based on IDEF1X, although heavily influenced by structured methods and Information Engineering. In this document I strive for a neutral, common-usage terminology and only occasionally use the formal IDEF1X terminology or notation.

The Data Modeler’s Skill Set

A data modeler, regardless of approach, will use the following skills:

Notation. Every modeler probably uses some version of the Entity-Relationship Diagram, although its importance relative to other tools can vary greatly. Although fundamental structural components are very similar, the notation used to express the model can vary widely. Most of the methodologies mentioned previously use some variant on the box for a data entity, with greater or less text detail visible. Relationships between the entities are shown with a line, and this is where the biggest notational differences show up. One Computer-Aided Software Engineering (CASE) tool that I use supports nine different notations for the ends of relationships (including arrowheads, crow’s-feet, IDEF1X) and permits the user to switch back and forth between them. They are all interchangeable, although sometimes you have to supplement the graphical notation with a written business rule to avoid losing some details. In the examples throughout this text I use arrowheads.

Naming Conventions. Modeling efforts often spend a great deal of time working out naming conventions for entities, attributes, and relationships. In my experience, it is more important to pick a convention and use it consistently than it is to agonize over the specifics of the convention. However, it is important to recognize physical limitations on names imposed by target implementation environments, at least during physical design stages. If a particular query tool ignores all but the first ten characters in a column name, don't name columns "CASE_DATE_OF_FILING" and "CASE_DATE_OF_APPROVAL".

Communications and Meeting Facilitation Skills.

Perhaps the single most essential skill for a data modeler is the ability to work with users and subject-matter experts to identify, document, and verify the business rules and logic that the data model must implement. A mix of large group meetings, small group or individual interviews, and textual research is usually required. The ability to communicate back to the clients in terms that they find easy to understand is essential. Facilitation is necessary to achieve consensus on data requirements, which can be a major effort.

Documentation Tools. Data modeling can generate enormous sets of documentation, especially if performed in the context of Enterprise Architecture Planning. Word processors, databases, CASE tools, and other tools for organizing and searching this material are essential support tools.

Definitions of Essential Technical Terms

Thus far we have freely used terms such as *entity*. Here we give a rigorous but not too technical definition for some of the key terms used through the rest of this document. We will attempt to reduce the level of technical jargon to a minimum, but it is difficult to present the data modeling process without some specialist terminology.

Although we do not stress the fact, creating any model component involves more than a simple declaration.

Each element must have a formal definition, and attributes must have format and domain specifications. Relationships have additional definition components such as cardinality. These should be completed to an appropriate level of detail as soon as a component is added to the model.

One useful metaphor for thinking about this process is *description*. Descriptions are about something (entities) and tell us something useful or instructive (attributes, etc.). A description is seldom exhaustive; instead it focuses on what we need to know right now.

What Is an Entity?

Data modeling involves making decisions about which aspects of the real world are of interest to us in solving a particular problem. These aspects may be things (e.g., people, concepts, events) or relationships between things (e.g., residence). These are real phenomena that can be represented in English by a noun or verb. We may need to represent a physical thing such as a building that can be touched or counted, or an abstract thing such as a ZIP code. Some things such as an address may be intermediate in nature—and may be treated as an entity for one purpose, but as an attribute for another.

We start modeling by organizing these aspects of the real world into logical classes—for instance, "George" and "Fred" and "Sue" can be put into many legitimate classes, but for purposes of an addressing database they each best fit into a class of "Occupant". For each class of things or relationships determined to be part of our system, we must create a corresponding representation within our database. This representation is the **Entity**.

(At this point a matter of terminology may be confusing to some people. Some data modelers prefer to use *entity* for a group—e.g., "Occupant"—and *entity instance* or *member* for a unique member of that group—e.g., "Fred". Others prefer to call the member an *entity* and the group something such as *entity set* or *entity class*. We will use *entity* to refer

to the group. This usage will give a straightforward, more easily understandable correspondence between the entity and the table.)

What Is an Attribute?

We included a category of things in our set of real-world aspects because we wanted to know some fact about it. One definition of a fact is a true statement of a property of the thing in question—a “good description.” Our concern in designing the database is to create a place to store these facts. Each fact identified as relevant is represented by a corresponding **attribute** in the data model. When we get to the physical design, the attribute will become a field in a database table (or some equivalent element).

It is vital to remember that we are only going to include relevant characteristics or properties of a thing. The statement, “The Southside Industrial Park is located at 123 Main Street,” specifies one property of that particular industrial park. Our understanding of the problem must guide us in selecting only those properties that are relevant from among the infinite number of properties that we might identify. For instance, the official site plan file for Southside Industrial Park might weigh 2.3 pounds, but we probably don’t care—unless our problem involves selecting file cabinets to store site plan documents! Because its properties are normally used to decide whether a particular thing is a member of an entity, most of the relevant properties can be used to describe all members of that entity. If not, this presents a possible means of refining our entity definition into two smaller classes, representing things that share this property and things that do not.

Attributes can be classified as “key” or “nonkey.” Nonkey attributes are easy to define: they represent those facts about the entity that do not meet the technical definition of a key. This is not very helpful, however, until we define a key attribute. We approach this task by distinguishing between two types of keys.

Primary Keys

A **primary key** is a set of one or more attributes that uniquely specifies a single instance of an entity. The role of a primary key is to find the desired member of the entity (or record in the table). If I know that Case Number is the primary key for site plans and that the value of the Case Number is “SP97-123”, then I can use the database to find the record for the Southside Industrial Park. The fact a primary key conveys about the member is identity, whether by name, identification number, or some other means. (Good database design encourages use of a “surrogate” key, such as an arbitrarily defined identification number. This is also known as a “dataless” key, though it may include some meaning such as sequence. Many commonsense candidates for a primary key, such as employee name, turn out to not be unique after all. Also, we want the primary key to be stable and unchanging—think about what has to happen when the Planning Department changes the way it assigns case numbers.)

Foreign Keys

A **foreign key** is an attribute that stores the value of a primary key in another entity. The role of a foreign key is to point to or find the desired member of the entity (or record in the table) that will complete a relationship pair with the member identified by the primary key.

The fact a foreign key conveys about the member is its relationship with some other member of either the same or another entity. For example, the record for Fred in the OCCUPANT table might contain an attribute called PARCEL, which for Fred is defined as “B2356”. When we query the physical database, we would read the data record from OCCUPANT where the name is “Fred”, then read the record from PARCEL where the PARCEL_ID is “B2356”. (This is an example of a relational join, and the ability to make such connections is the reason we go through all the rigorous effort of designing a relational database.) We would then find out that Fred is an occupant of the real estate parcel located at 566 Jones Drive.

Thinking about the contents of the OCCUPANT and PARCEL tables will help clarify the difference between a primary and a foreign key. If OCCUPANT is defined as containing information about all persons residing within an area of interest, then there may be many records where PARCEL equals “B2356”. After all, Fred is only one resident in his apartment building; each resident has an OCCUPANT record. Therefore, knowing the value of this foreign key does not get us to one and only one record in OCCUPANT.

However, think about the PARCEL table. Here, the PARCEL_ID attribute is a primary key. If we search this table for records where PARCEL_ID is “B2356”, we will find exactly one record—for the real estate parcel located at 566 Jones Drive, where Fred and the other residents of the apartment live.

Nonkey Attributes

As noted earlier, nonkey attributes represent all relevant facts about the entity that do not meet the technical definition of a key. They are essential to completely describe the entity members, but do not help to uniquely identify one member from another. For example, the user might query the database for all parcels that are zoned for apartments. The answer would include B2356, located at 566 Jones Drive, but it will also include many others.

What Is a Domain?

An attribute is a place to store a fact about each entity member. When we complete our physical database design, we will have defined each attribute as having a certain type and size of contents—for instance, the Street Type field may be stipulated as a “Character String” and may be up to four characters long. However, when we start putting data into this field, not all four-character strings may be correct even though the database can store them.

The **domain** of the attribute is a specification of the collection of valid values that a given property may

take. For example, a list of all valid street type abbreviations would tell us what could go in our Street Type attribute field. We can use this specification to make sure that, for example, “AVE” is not used to abbreviate “Avenue”. (In this example, “CALLE” will not fit and “QWER” will—but shouldn’t!)

A domain can take three types of specification (taken from the FGDC Content Standard for Geospatial Metadata, Version 2, Section 5.1.2):

- An **enumeration** is a list of valid values, such as a list of street types. If the candidate attribute does not exactly match one of the elements of the list, it doesn’t get entered.
- A **range** is specified by the first and last valid values within a larger, ordered list. For example, the domain may be defined as “all integers between 1 and 9”.
- A **codeset** is specified by naming a recognized set of values, any of which is accepted as a valid entry. (This sounds pretty exotic, but it is the default domain type for insufficiently specified attributes. In our street type example, without a list of valid values, any ASCII characters can be combined in one-, two-, three-, or four-character arrangements and stored—though only a very few would make any sense.) For example, an attribute defined as of the type INTEGER implicitly takes all valid integers from the largest negative value allowed by the DBMS to the largest positive value allowed, where size is determined by the number of bytes of storage allocated by the system to holding the value. This can be restricted to “all positive integers [that fit]”.

An attribute domain can be defined at any design stage where attributes are defined. Except for key attributes, this step typically is left to the physical design stage. However, some level of definition of the attribute is necessary for normalization, and domain specification is an excellent means of clarifying definitions.

What Is a Relationship?

A **relationship** represents a relevant association between two entities. If you will remember, we defined foreign keys by saying they identified the other half of a relationship. An association is any connection between entity members. The statement, “Developer XYZ filed site plan SP97-123,” is an instance of an association between two specified members. If this connection can be generalized to all (or at least most) members of the two entities, it can be directly captured in the database design.

There are two types of association, which are represented differently in our design. One type has no properties except that it associates members of two entities. This is typically represented as a logical relationship, and implemented by keys stored in both tables representing the members being associated. The other type has properties that cannot be associated with either entity but only with the association of their members. This will typically be represented as a third entity, plus a pair of relationships tying it to the two “primary” entities.

Defining entity relationships during data modeling does not require that every possible association between entities be identified. Rather, the relationships most significant to the user will be identified to validate the completeness of the entities and the user’s processes. There are three general types of relationships that we might encounter:

- Some attributes support the relational JOIN operation, but do not otherwise indicate any important relationship among the entity classes. A common example involves dates: within an area we can compare freight tonnage by year with annual farm production statistics solely on the date attributes. Some attributes potentially support a formal relationship, but are not sufficiently important to justify the effort involved in requiring the database management system to manage the relationship for us. (It is not necessary

to have a formal relationship defined to use the relational JOIN operator.) WARNING: This type of relationship may be structurally valid, but totally bogus—we can relate freight tonnage by year to employee year of birth, but why?

- Some attributes, however, are sufficiently critical to the semantics and syntactical integrity of the data resource that we insist the computer help us guarantee them. We do this by defining a formal relationship between the two entities. The DBMS technology may place varying restrictions on a relationship, but generally the relationship must have one-to-many cardinality and the attribute on the one side must have a unique index in place.
- Introduction of GIS technology to data management creates a third category of JOIN, using locational coordinates. This join is handled invisibly and automatically within the GIS engine, so we don’t have to worry about it. We do, however, have to specify our graphical data elements so that geospatial relationships evaluate correctly during operations such as overlay.

We have been talking about relationships as if they applied to every member of both entities. This is not necessarily true. The actual combination is referred to as the **cardinality** of the relationship. The various possibilities for combination between two entities are shown as follows:

Cardinality	Definition
One to One (1:1)	For each member in Entity A, this relationship must exist with one and only one member of Entity B.
One to Many (1:M)	For each member in Entity A, this relationship must exist with at least one member of Entity B. It may exist with more than one member of Entity B.

Many to Many (M:N)	For each member in Entity A, this relationship must exist with one or more member of Entity B, and vice versa.
One to Zero or One (1:0 1)	For each member in Entity A, this relationship may or may not exist with a member of Entity B. If it does exist, the relationship will exist with one and only one member of Entity B. (If Entity B is the parent relationship, this is referred to as a “parent optional” situation.)
One to Zero, One, or Many (1:0 1 M)	For each member in Entity A, this relationship may or may not exist with a member of Entity B. If it does exist, the relationship can be with only one or it may exist with more than one member of Entity B.

When you look at the cardinality of a relationship, it is very helpful to think of a pair of one-way relationships instead of a single bidirectional relationship. In the previous table, the specified cardinality is for one of these paired relationships. This approach allows us to discuss key-based relationships as being between parent and child entities, when the “parent” is on the singular side of the relationship. This terminology is most appropriate when we talk about identifying relationships, but has become commonplace for discussing any relationship.

IDEFIX includes an additional specification of the relationship, whether it is identifying or not. This defines how foreign keys will migrate to the many side of a one-to-many relationship. An *identifying relationship* is such that you cannot identify a record in the dependent relationship without first knowing the primary key of the parent entity. For example, a BUILDING can have one or many ENTRANCES. Typically, you will structure ENTRANCE to include BUILDING_ID as part of the primary key, rather than trying to create an identifier for improvements

that did not involve the parcel identifier. It makes no sense to think of entrances without a building. In contrast, a *nonidentifying relationship* places the foreign key as a regular attribute (which has a foreign key constraint).

The cardinality of a relationship is an important piece of information for database design. Certain types of cardinality are known to create problems in a database and should be removed. This information guides several of the actions taken during the normalization process in order to produce a well-structured database design.

Superentity/Subentity Relationships

The data modeling approach presented here utilizes a refinement of the basic Entity-Relationship Model that often seems to greatly clarify the data model. This is the concept of a relationship between a superentity (“super” in the sense of “above”) and one or more subentities.

For example, we may define an entity called IMPROVEMENT with which to represent the manmade structures on a parcel of land. Later in our modeling exercise, we might determine that we needed to distinguish between improvements in which people live or work and other types of improvements such as parking lots.

Our model could handle these by stipulating a complex set of relationships between the two entities. It makes more sense, however, to simply specify that members of BUILDING are a special case of IMPROVEMENT. This means that for each manmade structure we will collect the facts that are to be stored in IMPROVEMENT. If this improvement is habitable, we will also collect some additional information that is only relevant to BUILDING. (Note that we will have to carefully define what makes an improvement a building—for instance, at what stage of construction does a building become habitable?)

Putting the Pieces Together—How a Relational Database Works

The relational database definition specifies three types of operation that the nontechnical user would normally be aware of. Every user interaction with the database ultimately is reducible to a sequence of these operations. A `SELECTION` identifies records for specific members from a specific table. A `PROJECTION` identifies a subset of specific attributes from the set of attributes that makes up the record. A `JOIN` associates specific member records from two tables into one (or more) compound tables. Examples of these operations are:

Selection: “`SELECT * FROM STREET WHERE STREET_NAME = 'MAPLE AV'`” [Give me all attributes of all street segments named ‘Maple AV’.]

Projection: “`SELECT STREET_NAME, LOW_ADDRESS, HI_ADDRESS FROM STREET`” [Give me the address ranges for every street segment.]

Join: “`SELECT HOUSE_NUMBER, STREET_NAME FROM BUILDING, LANES FROM STREET WHERE STREET.STREET_NAME = BUILDING.STREET_NAME AND BUILDING.HOUSE_NUMBER BETWEEN STREET.LOW_ADDRESS AND STREET.HI_ADDRESS`” [Tell me the number of lanes in the street segment serving each building.]

Collectively, these operations provide the power of the relational database technology. This power justifies the effort required to complete a suitable database design. This power can be described in three broad aspects, as follows:

“Store Data Once, Use Many Ways”

The modeling effort results in an organization of data with virtually no redundant data. The only exceptions are the repetition of key attributes as foreign keys (the only way formal relationships are possible between tables) and a conscious choice by

the database administrator to risk certain operational problems to achieve performance goals. The data is accessible for many purposes, with no hard-coded constraints on use.

The Relational Join

The ability to join any tables within the database together when there exists a logical, meaningful relationship between them is what allows this reduction in redundant storage. (Strictly speaking, the requirement is only for a formal relationship, which does not have to be either logical or meaningful. I can “join” a table of employees to one of permits where `DATE_ISSUED` equals `DATE_BORN`, but why?) For example, you don’t have to store school district details with every record in your `SCHOOL` table. Instead, you store the name and address of the Superintendent of the Pleasant Valley Independent School District once in a `SCHOOL_DISTRICT` table, and use the identifier for Pleasant Valley I.S.D. as a foreign key in `SCHOOL`.

Furthermore, if no direct relationship between two tables exists, you can usually find a string of relationships that will ultimately let you establish the desired connection. This is called “navigating the database.” Remember that a join doesn’t require an explicit foreign key relationship.

The Geospatial Join

The introduction of GIS technology into the relational database arena allows a significant enhancement of the relational join. Now, not only can we join two tables through use of a common attribute, but we can also use the implicit attributes of location to join two graphical representations of entity members, based solely on their locational and/or topological characteristics. (This has some adverse impacts on data modeling, which are discussed in a later section.)

Why We Bother with Data Modeling

There are many “good” database designs possible in a particular instance, just as there are many “bad” ones. The only true measure of quality is whether or not the results meet the needs of the user. Unfortunately, this is difficult to measure prior to implementation, so it tends to be a “Mom and apple pie” kind of goal. We can, however, describe certain problems in a database design that will almost certainly lead to failure to meet the users’ needs.

Problems of Appropriateness in Design

Is the problem adequately defined? “Why can’t we retrieve that?” “But we need that sorted by size of project!”

Does the data fit the problem? “The questions you’re asking are about buildings but the data is collected about parcels!”

Do pieces of data that are intended to be associated actually represent the same thing? “We can’t match this data about subdivision lots with that data about parcels.” Until the last ordinance change, the two had no relationship with one another—I own a parcel comprised of Lot 1 and the eastern five feet of Lot 2 in my (late 19th century) subdivision.

Can pieces of data that are intended to be associated actually be associated? “The only common data between the utility hookup file and the Assessor’s file is address, but we can only get a 10 percent match when we try to do a join!”

Problems of Definition

Synonyms in Data Dictionary. “This file contains an attribute labeled ‘CASES’ but this other file contains one labeled ‘APPLICATIONS’. Are these the same thing?”

Confused Referents. “I want to match on these two files. Does NAME refer to applicant or to developer

in this file?” “Is this coverage for tax parcels or for lots?”

Incompatibility of Data Formats. “ZIP code is stored as a number in my file, but as a ten-character text field with a hyphen in that one. How can we match on these fields?”

Problems during Implementation

Storage Anomalies. Redundant data is stored in several places. Practically speaking, this may or may not be a problem. It may be inefficient to use the extra storage, but with frequently asked queries this may be offset by the increased efficiency of response. It may also create inconsistencies in the database, a potentially serious problem, when transactions that modify database contents do not modify all storage locations of redundant data.

Update Anomalies. If data is stored redundantly, it is possible to update data in one location but not in another. Such potential inconsistencies are especially hazardous in attributes used as keys or as selection criteria in a query—which is why we want keys to be unchanging. One of the best arguments for surrogate, dataless primary keys is that you never have to change them once they are assigned.

Insertion Anomalies. Because of the design of the database, we may not be able to store all the data items we desire. For instance, if a developer’s name is stored as part of a project record, how can we store information about a developer with no currently active projects? (Of course, the designers must ask themselves: “For the purposes of this application, do we really care?” This is only a problem if it prevents you from meeting a project objective.)

Deletion Anomalies. Because of the design of the database, we may not be able to preserve needed data items when other data is deleted. This is the inverse of an insertion anomaly. For instance, if a developer is stored as part of a project record, how can we

continue to store information about a developer who completes his or her last active project?

Missing Values. There may be no data available to store, even though a place has been reserved for it. This can take two forms, one of which is relatively trivial in its design implications. This happy state of affairs occurs when certain data items are optional for those entities represented by the record in question. For example, a land-use record may contain a field for SIC code; however, only about a third of the possible uses can be assigned a valid SIC code. Here, a blank probably means “Code does not exist.” (In this example, the designer might consider moving SIC code into another file to free up the storage allocated for records with missing data.)

However, there may be more serious problems caused by missing values. If “blank” is a legitimate value, how does one distinguish missing values? This is most critical in software that sets the default value of numeric fields to zero; injudicious use of this value will skew any averages or other descriptive statistics using that field. This problem can occur in any

instance where one wishes to distinguish between “none” and “unknown”.

Problems of Correctness

Formal Correctness. Are there technical errors in the structure of the model, such as a relationship that is not anchored to an entity at each end?

Correct Representation of Reality. Does the model contain true statements about the structure of those aspects of reality chosen to be relevant for the enterprise or problem? For example, are all the real-world things (e.g., buildings, signs, utility service) to which an address can be assigned are represented in the data model?

Correct Representation of Business Rules. Does the model contain true statements about the business rules of the enterprise? For example, are addresses only assigned to buildings, or can an unimproved parcel get an address?



SECTION 3

PERFORMING DATA MODELING

This section begins with a systematic process for performing data modeling, which includes a greatly simplified example that is worked through as part of each stage. The technical issues of normalization are discussed separately.

A Process for Data Modeling

In this section, we provide a step-by-step overview of performing a data modeling project. The terminology and concepts introduced previously are integrated into the discussion. The stages of this process correspond closely but not exactly with those prescribed in IDEF1X, but the presentation is intentionally methodology-neutral.

The database design methodology that I am presenting distinguishes four stages in the complete database design life cycle. My experience is that almost all methodologies have the same general sequence of events, but that different practitioners have preferences in tools used and the breakpoints between the recognized stages. The CASE tool used relies primarily on an Entity-Relationship Diagram and associated data dictionary, though it supports many other tools and techniques. For example, in designing a geoda-

tabase as specified by ESRI's ArcGIS 9, we switch to an object-oriented static structure diagram based on the ArcObjects template for physical design and implementation planning. The following discussion briefly summarizes this conception of the database design life cycle, so that our proposed approach to this task can be clearly understood.

This discussion also assumes that both project management disciplines and prerequisite studies of needs, work flows, etc., are completed and available to the modeling team. Reality is seldom so cooperative, so a data modeler will of necessity have significant interactions with teams performing these and other activities. (See other documents within this *Quick Study* series for guidance on different aspects of IT projects.)

Conceptual Design—Defining the Model's Scope

Define project scope. The first stage begins by defining the scope of the data modeling project. This may be already defined within the context of a larger IT project, but it is essential that the business scope be clarified and then converted into a high-level

data model for verification. Regardless of how these are generated, the data modeling team must have adequate performance specifications (“design-to” requirements) to determine what must be included in the model.

The business scope of the project tells us what the business problem(s) is we are trying to solve. For an explicitly enterprise project, we need to know what the mission of the enterprise is. This context not only helps us determine the contents of our model in subsequent steps, but also has a significant impact on what kind of data model we are trying to build. A data model to support a tightly focused project is relatively straightforward, while a decision-support project may require a data-warehouse approach.

The issue is confounded by what I call the General Systems Paradox, which says:

Everything is related to everything else.

Every system is a subsystem of some other system.

Every system is comprised of some other systems.

Given this complexity, how can we do a complete data model? Because we can’t include everything, what do we include? How do we choose?

The way out of the paradox is simple. We are trying to solve some problem. From that problem perspective,

*Everything is related to everything else,
but some relationships **matter more** than others!*

We define the model’s scope using the following process:

Collect business documents that are clearly relevant to the problem. These might include mission statements, enterprise vision, enabling legislation, project charter, technical reports, maps, etc. In a larger IT study, existing data and reports are obvious starting points.

Identify candidates for representation. Highlight nouns, phrases, and verbs that imply transformations in the selected documents as keywords. Organize these into a hierarchical list of topics, with consolidation into subheadings whenever possible. Polish up the language for consistent formats and levels of detail, then make whatever adjustments to the hierarchical list seem appropriate. Don’t worry about discarding details at this stage, but keep notes that may help you flesh out the more detailed models to be developed later.

Create a conceptual data model. This level of data model is intended to provide a high-level overview of the database. In this overview, data elements are grouped in general clusters that reflect the external, business-oriented perspective on the data. The conceptual model constitutes a formal statement of understanding of the major data components of the problem, of the critical business relationships and rules. It also serves as a declaration of scope—data components not included in the conceptual database design are excluded from the project scope unless a clear need for inclusion is later demonstrated.

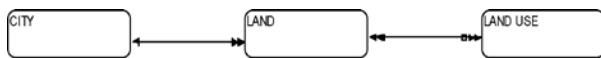
This model corresponds to the Entity Model in IDEF1X (also, confusingly enough, called an “Entity-Relationship Diagram”). To create an entity model, use the ERD technique (see Section 2.5 for details) to create entity symbols for the highest-level nouns from our topics list. Examine lower-level terms from the listing to see if they require a subentity representation. The major relationships among these should be sketched in at this time, but without much rigor. The level of definition of model contents that is needed at this stage is minimal—primarily, the name of the topics. In fact, I frequently will stop with a simple list of candidate entities, with minimal definition.

Review and refine model with users. The resulting model should be tested against the mission statement and other source documents, basically to see if you can recognize the enterprise by looking at the data model. The same question should be posed to key

users. The model should also be checked to see if it hangs together, without any gaps in logic or islands of data. Finally, it should be checked against the project mission to make sure we are answering the right questions.

An Example—

The Mayor says, “We need to know how the land within the city is being used.” We ask a few clarifying questions and decide we have the following candidate entities: LAND, CITY, LAND USE. Our conceptual model therefore looks like this:



We review this with the Mayor’s assistant and discover the following clarifications:

- The Mayor is interested in the tax base within certain areas of the city.
- There is no interest in any land lying outside the city.

The revised conceptual model looks like this:



Logical Design

The conceptual data model is the input into a more detailed and rigorous modeling process that produces a logical data model. This process concentrates on a clear definition of the conceptual design entities and the formal specification of process rules as relationships. The resulting design is focused on the perspective of the user. The more important attributes of each entity are defined, specifically including primary and foreign keys and sufficient attributes to support normalization. The technical rules and procedures of database normalization are applied to the evolving design in an iterative manner until an appropriate degree of normalization is achieved, to produce a

design that avoids certain types of possible error conditions. The resulting design has well-structured and well-understood entities and relationships, with most attributes defined.

In my practice, I find the “best” results from logical design occur when we deliberately ignore two possible design constraints:

1. How can we implement this?
2. Where will we find the data to populate it?

The reason to ignore these at the logical design stage is simple. If we need to know it, then we need to know it. Period. An inability to deliver the data the users need does not eliminate their need.

Ignoring these constraints, however, has two strong impacts on subsequent stages of design. First, inclusion of a data element in the logical model cannot be taken as a commitment to deliver it in a physical implementation—if soils data is simply not available, the implementation team cannot be faulted for not magically finding it somehow. Second, inclusion in the logical model is a statement of priority—if soils data is needed but not available, then (if it is important enough) its acquisition should be considered during next year’s budget deliberations.

Break up the entity clusters. The high-level entities in the conceptual database design are now decomposed into more refined and specific entities. For instance, the conceptual entity “jurisdiction” might generate logical entities “city”, “county”, “administrative district”, etc. This process concentrates on a clear definition of the entities. This defines a set of logical entities that directly relate to business practices, and helps clarify the primary relationships among them. Most of the candidate entities from our earlier listing should be included at this stage—in one form or another.

Establish well-defined relationships. This process concentrates on a formal specification of business

rules as relationships. Each relationship at this stage of modeling has two characteristics: (1) It completely represents business rules regarding possible relationships (“An ADDRESS can only be assigned to a BUILDING”) and (2) It represents business logic that will be enforced by the database implementation through foreign and primary key constraints (or the equivalent, if the implementation is not going to use a relational database management system). This means that not all business rules can be explicitly included within a data model; some will only be documented as implementation notes to be accommodated within application logic. Also, if any many-to-many relationships still exist within the data model, these must be resolved at this time.

Create key-based model. The components are now assembled into a unified entity-relationship diagram. The resulting design is focused on the perspective of the user, and ignores all implementation issues. The more important attributes of each entity are defined, specifically including primary and foreign keys. This corresponds to the key-based model in formal IDEF1X methodology. It should be tested to ensure that the rules of Entity-Relationship Modeling are satisfied, and that the model appears complete.

Normalize the key-based model. The technical rules and procedures of database normalization are applied to the evolving design in an iterative manner until an appropriate degree of normalization is achieved. The resulting design has well-structured and well-understood entities and relationships, with some attributes defined. The intention of the normalization process is to produce a database design that can be implemented without creating errors or problems during use. Normalization of a key-based logical model will of necessity be incomplete, for not all attributes are defined at this stage of modeling.

An Example—

We review the entity clusters in the conceptual model for possible refinement. We captured all the nouns

on our initial list, but the review indicated that we do not need CITY after all. Also, PROPERTY is better named as PARCEL. At the end of the review, we have the following:

PARCEL. A taxable unit of land within the city limits. “PIN Number” is the attribute that identifies a particular parcel.

LAND USE CLASS. The human utilization or type of behavior that occurs on a PARCEL. This is distinct from land cover, which would be the vegetative or other observable surface features occurring on the parcel without regard for their human utilization. “Land-Use Code” is the attribute that identifies a particular land use.

The relationships among our entities can be specified:

A Parcel will always have at least one land use occurring on it, since “Vacant” is explicitly defined as a valid land use. However, it may have many land uses.

A land use may be recognized but not at present occur within the city.

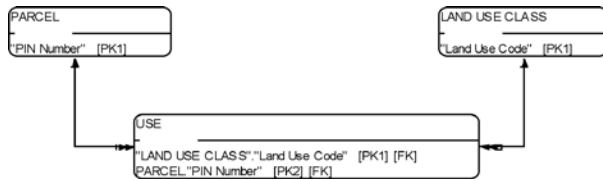
The initial key-based logical model therefore looks like:



Our first review identifies the existence of a many-to-many relationship between PARCEL and LAND USE CLASS. Because we are going to work at a key-based logical design level, we cannot allow this to remain because there is no place to put the foreign keys defining the relationship.

Following standard normalization practice, we replace the many-to-many relationship with a third,

associative entity called USE. We link it to the two original entities with a pair of one-to-many relationships, with the associative entity on the “many” end of both relationships. This allows migrating the primary keys of PARCEL and LAND USE CLASS into USE as foreign keys. Our revised model therefore looks like:



Fully Attributed Model

Designate who adds attribution to model. We are now getting to the point of demarcation between data modeling and database design. I try to make an explicit and clear break between data modeling and physical database design activities, and another between database design and implementation. The functional reason for these distinctions lies in the shift from business logic to technical implementation logic required to begin database design, and from technical logic to the site-specific knowledge of storage arrays, network and server architectures required to plan and perform implementation at the client site.

(This discussion proceeds as if the data modeling team will perform all stages of the implementation. The reader is encouraged to adjust this suggested implementation technique to fit his or her situation. Certainly, the later stages of work will require services of one or more database administrators [DBAs], whose hands-on technical skills with implementation technology [e.g., Oracle, DB2, SDE, et al.] becomes essential.)

This step can constitute either the last stage of data modeling or the first stage of database design. If I am stopping the modeling exercise prior to physical database design, I will complete the fully attributed

logical model as the final deliverable. However, if I am also doing the physical database design, I will typically treat this stage as the first activity in that stage of the project. It is best if this decision were made at the beginning of the modeling project, but sometimes circumstances require that we decide only at this stage of work.

Add all attribution to the key-based model. All nonkey attributes that have been identified during the modeling process should be added to the key-based model at this time. The results will correspond to the Fully Attributed Model in IDEF1X. Each entity should now have every attribute assigned that we wish to capture for it.

Normalize the fully attributed model. The technical rules and procedures of database normalization are again applied to the evolving design in an iterative manner until an appropriate degree of normalization is achieved. Normalization of a fully attributed logical model will by definition be complete, for both all key attributes and all nonkey attributes have been defined at this stage of modeling.

An Example—

We continue with the modeling exercise. We gather additional information from our users to complete the attribute definition for our model. The results are summarized as follows:

ENTITY	ATTRI-BUTE	KEY	DEFINITION
PARCEL	PIN Num-ber	Primary	Parcel ID Num-ber assigned by Tax Assessor.
PARCEL	Address		Situs address as-signed to parcel or building(s) on parcel.

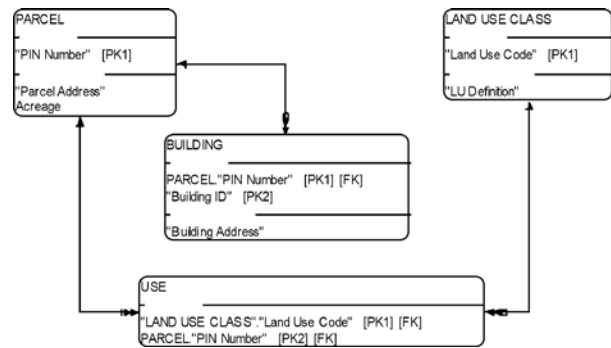
PARCEL	Acreage		Official area of parcel in acres, to nearest hundredth of an acre. Determined from legal instrument, e.g., subdivision plat.
LAND USE CLASS	LU Code	Primary	Three-digit code for land-use classification, as used by Planning Department.
LAND USE CLASS	LU Definition		Definition of land-use classification, as used by Planning Department.
USE	PIN Number	Primary; Foreign	Parcel ID Number assigned by Tax Assessor. FK from PARCEL.
USE	LU Code	Primary; Foreign	Three-digit code for land-use classification, as used by Planning Department. FK from LAND-USE CLASS

Normalizing this model reveals several problems, including:

- The attribute PARCEL.ADDRESS has two problems. First, it can be a repeating group if two or more buildings on a single parcel have different addresses (violating First Normal Form). Second, the address is not always functionally dependent on the PIN Number, but on some combination of the PIN Number and the building (violating Third Normal Form). Actually, these are two manifestations of the same problem and can be dealt with at the same time.

- Is land use functionally dependent on the PIN Number, as indicated in the relationship between USE and PARCEL? Alternatively, should it be related to buildings in some manner? (GIS users will recognize this as a variant on the question of whether we need to indicate where within the parcel the land use occurs or if it is sufficient to simply indicate that it occurs somewhere within the parcel.)

After consultation with our client, we defer some of the decisions until the physical design. The results after normalizing the address problem are:



Physical Design

The biggest difference between data modeling and database design is the shift in orientation to the problem. A logical model focuses on the business perspective, with no concern for how the model might be implemented. Now we explicitly shift our orientation to the computer.

The logical model is now assessed from a perspective of (machine) implementation. Decisions are made as to which type of dataset each entity will be represented in—for instance, deciding if the PARCEL data should be an Oracle table, an SDE layer, or both. The final record layout is specified and the normalization

process applied to the design again. Any remaining design issues are dealt with at this time, as are performance concerns that might encourage a degree of denormalization. The list of process rules is checked for database implementation, and if any are found to be missing they are specified in integrity constraints, triggers, or other stored procedures as supported by the RDBMS. The results are captured in a mix of diagrams and tables, and contain all the structural information necessary to develop a physical implementation plan.

Commit to an implementation architecture. The fully attributed logical data model is assessed from a perspective of possible machine implementations. Decisions are made as to which type of dataset each entity will be represented in—for example, deciding if “municipality” data should be an Oracle table, an Arc/Info coverage, or both. Hopefully, the possible choices for implementation have already been defined for the database design team. If not, the project should go on hold until these complex and challenging decisions are completed.

The database design team, possibly in concert with the applications development staff, now must pick from among the possible implementation technologies the ones to be used for this implementation of the data model. Remember, the three-schema approach permits the conceptual schema to be implemented in multiple ways so long as each physical implementation schema correctly represents its subset of the conceptual schema. The database design team, however, must decide what tools its particular physical schema will be implemented in.

Select portion of logical schema to be implemented. The physical database design team is not required to implement the entire schema embodied in the fully attributed model. That subset of the model associated with a particular business process or other theme may be separated from the rest for an implementation—or instance, utility infrastructure data may be segregated for a facilities management system, or the

details of land-development cases for a permitting system. Typically, an enterprise data model will be implemented in multiple data repositories, while a process-oriented data model will be implemented in a single repository.

Transform logical schema into physical database design. The final record layout for the chosen implementation technology is specified. Typically, an entity will become a database table, or GIS layer. Attributes associated with the entity become columns within that table or layer. The relationship logic is implemented using the tools the data management system provides, such as integrity constraints and other stored procedures provided by an RDBMS. I find that the more design details I incorporate into the logical model, the closer the physical design will correspond to it. Conversely, any design details not resolved in logical modeling must be dealt with at this stage.

Even if we decide there should be a one-to-one equivalence between logical data elements and physical data structures (that is, the logical PIN Number will correspond to the physical database column PIN), we have several design decisions to make:

Do we store the data value as is or modify it in some manner? The PIN Number is created by the land-registration and taxation processes. Our first decision is whether we want to use this attribute as a primary key or use it to generate a suitable key value. If the PIN is unique, not subject to change over time, and not subject to reuse for a different parcel at a later date, then it might be fine as a primary key. Otherwise, we would record it as is as a nonkey attribute and add a surrogate PARCEL_KEY attribute to the PARCEL table. Less drastic modifications might include the insertion or removal of punctuation (e.g., the hyphen in a ZIP+4 postal code), standardized spelling, and capitalization, etc.

What kind of data type should we use to hold the data element? The PIN Number is created by the

land-registration and taxation processes. We need to examine a list of sample values to decide how to store it. If the PIN sometimes contains alphabetic characters (e.g., “B2356”), then we can’t use a numeric format; we would have a data type mismatch and find that the database wouldn’t let us store a valid value in the attribute. The Assessor assures us that the PIN is always a numeric value, so we can use a numeric data type for storage.

How much storage space do we need now and to accommodate future growth? The biggest PIN on the list the Assessor gave us has six digits. We know there are approximately 100,000 parcels of land in the city. We could therefore feel comfortable reserving six digits of storage—a Numeric (6) format in our examples—which would hold parcel PINs up to 999,999. However, we might want a larger storage value because (a) there are a number of parcels being split up for more intensive development, (b) many older buildings are being converted into condominiums that get separate PINs, and (c) we’ve been told there may be a countywide reassignment of PINs in the next year or so. If we have reliable growth estimates (e.g., 100 new cases a year), we can calculate a realistic storage size. If not, a convenient rule of thumb is to take the best estimate of sizing and add one additional significant digit (e.g., if we move the PIN attribute up to Numeric (7), we now have (literally) an order of magnitude increase in the maximum permissible value from 999,999 to 9,999,999!

Is this attribute mandatory? If we want to insist that every parcel record has a PIN number we make this mandatory, usually specified as “NOT NULL”. If, however, we recognize the timing of new parcel creation is such that the Planning Commission will create a parcel several months before the Assessor assigns a PIN, then we need to specify this as optional. (WARNING: If the PIN is optional, then it cannot be a primary key! We would have to modify our design as described previously to make the PIN a nonkey attribute that accepts NULLs and create a PARCEL_KEY to serve as a primary key.) The

obvious compromise won’t really help—we could specify the PIN as NOT NULL but define “blank” as a valid PIN value, but the PIN still could not serve as a primary key because more than one parcel would almost certainly have “blank” as the value of its PIN number at the same time. And to be a primary key, an attribute must be unique for each record!)

The results are captured in a mix of diagrams and tables, and contain all the information necessary for a physical implementation plan to be developed. This design corresponds to the Transform Model in IDEF1X.

Test and refine the design. Several design issues may need to be dealt with at this time. Many people will have performance concerns that might encourage a degree of denormalization. These concerns must be addressed, though preferably not through denormalization. The fundamental criterion for accepting a proposed denormalization is to ask if it changes the business rule. If it does, it should be rejected—better a slower correct answer than a fast but wrong answer. The list of business rules is checked for database implementation, and if any are found to be missing, they are specified in integrity constraints, triggers, or other stored procedures as supported by the RDBMS.

An Example—

The questions we had postponed about how land use should be related to parcels were again discussed with the Mayor and her staff. She decided that she did need to know where on the parcel a land use was located. In turn, this led our design team to the realization that the database would have to include a GIS component. The city GIS standard is “Brand X,” which supports storing and managing both graphic and nongraphic attributes. Therefore, the implementation architecture will be Brand X on the Mayor’s aide’s desktop machine. Furthermore, the physical design will deal with the complete logical model in this situation.

The transformation of the logical model into a physical model required no changes to the ERD. Although some CASE tools can generate other diagram formats

that can sometimes add clarity to the physical model, it is not necessary to do so here. The resulting specification is shown in the following:

BUILDING

DEFINITION: A habitable manmade structure. It may be intended to be used as a residence, a place of economic activity, or for some combination of human activities.

IMPLEMENTED AS: Brand X geospatial layer

BUSINESS TABLE NAME: BUILDING

FEATURE TYPES: Polygons

Layout for BUILDING

Item Name	Description	Key	Format	Null	Domain
PIN	PIN number	PK, FK	Numeric (9)	N	Assigned by Assessor to PARCEL
BLDG_ID	Dataless key	PK	Numeric (3)	N	Identifies a specific building
BLDG_FOOTPRNT	GIS-created graphic attribute.		POLYGON	N	
BLDG_ADDR	Situs address for building (if any)		Character (40)		
BLDG_SQFT	Size of building		Numeric (8,2)		

PARCEL

DEFINITION: A taxable unit of land within the city limits.

IMPLEMENTED AS: Brand X geospatial layer

BUSINESS TABLE NAME: PARCEL

FEATURE TYPES: Polygons

Layout for PARCEL

Item Name	Description	Key	Format	Null	Domain
PIN	PIN number	PK	Numeric (9)	N	Assigned by Assessor
PARCEL	GIS-created graphic attribute.		POLYGON	N	
PRCL_ADDR	Situs address for parcel.		Character (40)	N	
PRCL_ACRES	Official acreage for parcel		Numeric (8,2)		

LAND USE CLASS

DEFINITION: The human utilization or type of behavior that occurs on a PARCEL. This is distinct from land cover, which would be the vegetative or other observable surface features occurring on the parcel without regard for their human utilization.

IMPLEMENTED AS: Brand X lookup table.

BUSINESS TABLE NAME: LU_CLASS

FEATURE TYPES: N/A

Layout for LU_CLASS

Item Name	Description	Key	Format	Null	Domain
LU_CODE	Land-use abbreviation	PK	Character (3)	N	Defined by Planning Department
LU_NAME	Land-use name.		Character (40)		Short name for class
LU_DEF	Definition of land-use class.		Character Varying (255)		Definition of land-use class

USE

DEFINITION: The land use that occurs on a parcel of land.

IMPLEMENTATION NOTE: 1) Originally, this was an associative entity defined to break up a many-to-many relationship between LAND USE and PARCEL. Once it was determined that the location of land use within a parcel was required, it became a geospatial layer.

IMPLEMENTED AS: Brand X geospatial layer

BUSINESS TABLE NAME: USE

FEATURE TYPES: Polygons

Layout for USE

Item Name	Description	Key	Format	Null	Domain
PIN	PIN number	PK; FK	Numeric (9)	N	Assigned by Assessor to PARCEL
LU_CODE	Land-use abbreviation	PK; FK	Character (3)	N	Defined by Planning Department for LU_CLASS
PARCEL	GIS-created graphic attribute.		POLYGON	N	That portion of the parcel that predominately is in the specified land use

This design was reviewed with the clients and tested. The following supplemental business rules were discovered and added to the technical documentation. These did not require changes to the data model, but provided formal statements that justify the design.

- A situs address may be assigned to a vacant parcel or to a building. If a parcel is initially assigned an address while vacant, and a building is subsequently added, then both the parcel and the building will have addresses. These addresses are not necessarily the same.
- A building may or may not have a situs address assigned to it. A parcel will always have a situs address.
- Land use describes a portion of the earth's surface, regardless of the presence of buildings or other structures. A building may help delineate the boundary of that area.
- The location and extent of land use within a parcel is of primary importance. Generating multiparcel land-use maps is of secondary interest only.

Physical Implementation

Plan the implementation. The physical design must still be turned into an operational database. This is often the point at which a designer turns the process over to a database administrator (DBA), because of the detailed site-specific and tool-specific technical knowledge required for this planning. The DBA develops an implementation plan that determines disk space requirements, the location of each component, indexing, and related highly technical issues that are beyond the scope of this study.

Code the DDL. Data Definition Language (DDL) scripts or equivalent technical specifications must be developed to create the database, create each table in the database, define each field and index for the table, and implement any stored procedures. This is frequently performed using SQL, but a proprietary

scripting language may be used. Some newer RDBMSs provide a GUI, forms-driven administration utility that may be used instead of a batch script. However, most experienced DBAs prefer the batch script for the initial creation of a database instance.

Implement the data repository. Finally, the data repository must be built, loaded, and tested. This involves a number of implementation-specific issues that are beyond the scope of the data modeling activity. Ultimately, however, the actual implementation of the data model is an essential checkpoint of modeling quality.

An Example—

(Because we do not have the technical manuals for a Brand X database, we will defer completing our example. Instead, we will provide the DBA with a copy of these specifications and support them in actually setting up the database for the Mayor.)

Normalization and Denormalization

Normalization is the process of systematically removing from the database design the anomalies identified in earlier sections of this document. It represents probably the most daunting aspect of database design for the uninitiated, although, in fact, it is usually not all that difficult in practice if the preliminary analyses were adequate. “The normalization theory is based on the observation that a certain set of relations has better properties in an inserting, updating, and deleting environment than do other sets of relations containing the same data.” (Atre 1980:131)

The term originated with the initial definition of the relational database architecture, and is almost exclusively restricted to that model in usage. The concepts, however, are relevant to almost any logical design problem. Various writers have found new normal forms since the original description. Because of the way these are defined, most new forms are an extension of previous forms. Thus, any relation in Second

Normal form is by definition in First Normal; however, not all relations in Second Normal are in Third Normal form. The Domain-Key Normal form (Dutka and Hanson 1989) appears to place a limit on this extension—no more restrictive form appears possible. However, this does not prevent the definition of additional normal forms between Fifth Normal and DKNF, as researchers identify new problems and specify normal forms that eliminate them.

The following table summarizes the major, accepted normal forms:

NAME OF FORM	DEFINITION
Unnormalized	Cannot be expressed as a “flat file.” Contains groups of repeating data.
First Normal	Flat file with no repeating data.
Second Normal	Every attribute is either part of a key or is fully functionally dependent on a key.
Third Normal	Every attribute is either part of the primary key or is fully functionally dependent on it. No nonkey attribute is transitively dependent on a key.
Boyce-Codd Normal	Every determinant is a candidate key; primary key implies all functional dependencies.
Fourth Normal	Every multivalued dependency is a functional dependency.
Projection-Join Normal	The set of key dependencies guarantees relations have capability of lossless joins.
Fifth Normal	Join dependencies are satisfied.
Domain-Key Normal	All constraints are implied by the key dependencies and the domains of the attributes.

In practice, most modelers are satisfied to get a data model to Third Normal form.

The Normalization Process

Normalization is often referred to as *normalization by decomposition*. It consists of eliminating problems in an entity by breaking it up into smaller, simpler entity specifications. This supports the modeling concept that every entity should represent only one type of real-world thing. Various approaches to normalization exist, some of which bypass several individual forms. The following is a “quick-and-dirty” guide to performing normalization. The interested reader is referred to Brackett (1990) for a more complete but still business-oriented guide to normalization. Dutka and Hanson (1989) provide formal definitions and more algorithmic approaches based on the mathematics of sets. For an alternative approach, see Finkelstein’s discussion of *business normalization* (1992).

The criteria for determining whether a given specification is a problem depend entirely upon the analysis of the real-world situation that we are trying to represent in our database. Given an entity CASE with three status fields, if we are certain that every case will have at most three statuses that we want to store, then we probably do not have a problem in database definition. If we are also certain that the majority of cases will, in fact, have exactly three statuses, we are even safer. If, however, the number of statuses can vary between one and some fairly large number, then we have repeating data and a definite design problem.

Normalization can be applied to some degree to a data model at any stage of development. It is most complete when performed on a Fully Attributed Model. It is even better if all entities and attributes are fully defined, and domains specified for all attributes, as in a completed physical database design. Normalization involves aspects both of syntax (the structure of the model components) and semantics (the meaning of the components and the values to be stored therein).

1) **Review relationships.** The first step in normalization is a review of problematic relationships in the model. This includes:

Remove nonspecific relationships. If any nonspecific (i.e., many-to-many) relationships still exist within the model, remove them at this time. A many-to-many relationship is not implementable within the relational database architecture and should be replaced. Create an associative entity that represents the business logic of the original relationship. Insert it between the two original entities and connect them to the new entity with a pair of one-to-many relationships. (See sequence of illustrations in the example given in Section 3, above, for diagrams illustrating how this works.) The associative entity will have a compound primary key, which consists of migrated primary keys from both of the original entities.

Consider collapsing or restructuring one-to-one relationships. A one-to-one relationship is frequently (but not always) a sign of incomplete or misunderstood business logic. First, review the business logic to ensure completeness. Then if necessary replace the relationship. Common wisdom says that mandatory one-to-one relationships describe two aspects of “the same thing” and should be collapsed into a single entity. Optional one-to-one relationships often can be resolved by replacing the pair with a superentity/subentity structure in which the single entity has two subcategories, one with certain characteristics and one without. Some modelers will insist that a one-to-one relationship is the best way to represent business logic, but they should be prepared for some difficult issues during implementation, especially if there is any time lag between entering new entity instances between the paired entities.

Consider connecting isolated entities. Traditional data modeling expects every entity to be connected to the rest of the data model by at least one relationship. If not, common wisdom holds, why does it belong in this model at all? This assumption is not technically sound, because, as previously discussed, a relation between two entities does not require an explicit relationship to be implemented, but it does raise a useful red flag for further review.

Both enterprise models and GIS models frequently challenge this convention. Enterprise models may include islands of data that correctly reflect inadequate cohesion among multiple lines of business. GIS data is even more problematic—a data layer such as FENCE may have no direct logical relationship with anything else in the model, yet be desired for “context” in displaying other data layers. The problem is compounded by the explicit intent of a relationship to support the referential integrity through the migration of foreign keys.

A modeler confronted with isolated entities should first confirm that a relationship has not been overlooked that would connect the isolate. There are two approaches to resolving the isolation, if it cannot be ignored. (Some CASE tools will object to an isolate, and will sometimes give erroneous diagnostics when built-in error checking is performed.) First, a spurious relationship and possibly an equally spurious entity can be created at logical modeling stages, with the explicit intent of eliminating these before or during physical implementation planning. For example, earlier in my career my logical ERDs often had spurious nonspecific geospatial relationships among GIS layers (identified as “GISxx”) in order to fool the CASE tool I was using. These are different from true relationships because they rely only on coordinate attributes to establish the relationship. These can be dropped because, in effect, the feature overlay functionality of a GIS generates the associative entity on demand using the coordinates of the entity instances in question. Readability of the model requires that only a few of the possible geospatial relationships be inserted, since each GIS layer entity has the same kind of georelational relationship with every other GIS layer entity. Alternatively, the modeler can split the data model into two or more internally consistent submodels.

Look for relationship cycles or other structural problems. A loop of relationship dependencies—where A is in a one-to-many relationship with B, B is in a similar relationship to C, and C is in a similar

relationship to A—cannot be implemented because primary keys cannot be propagated without creating an endless loop. This situation implies incorrect logic. First, check to see if the model actually captures the business logic. Often this situation occurs when a relationship is inverted during modeling. If the model corresponds to the business logic, probe your customers to make sure you heard them correctly. If the looping dependencies persist, look for a restructuring that preserves the business logic that is implementable. (Reingruber and Gregory 1984 provide several examples of such structural problems as well as techniques for resolving them.)

2) Review entity structure. The following steps are performed for each entity in the data model. For convenience, the modeler should look for a focal point in the model and work outwards. A focal point is an entity that either has numerous relationships with other entities or exists only on the “one” side of every relationship it participates in. PARCEL, for example, is a common focal point in land-development databases. For each entity in the data model:

Verify the keys. Make sure the entity is well structured, with a designated primary key and at least one nonkey attribute, which may possibly be the geometry, as in my FENCE example when all we care about is the existence of the fence but not its height or construction material. If relationships require foreign keys to be included in the entity, confirm that these are correct. Also, check that the primary key has been propagated into other entities as required. (Technically, an entity does not actually require a specified primary key. At implementation time, the RDBMS or other tool can deal with the absence of a key through internal, hidden data elements. However, lack of a primary key can cause significant performance problems and in some tools can prevent implementation of an explicit relationship. I always declare a primary key.)

Check for First Normal form. An entity in First Normal form can be stored in a flat file. That is, there are

no repeating sets of attributes. To eliminate repeating data, we decompose the initial entity into several new entities, so that each repeating set of attributes is in a separate entity. We then duplicate the primary key of the original entity as part of a compound primary key for each of the resulting entities. This key duplication is accomplished by creating an identifying one-to-many relationship between the new case entity “B” and the new case status entity “C”.

Let us start with an entity A (CASE <pk>, APPLICANT, STATUS1, DATE1, STATUS2, DATE2, . . .). The results of converting to First Normal form are: B (CASE <pk>, APPLICANT) and C (CASE <pk, fk>, STATUS, DATE). (There are some problems with C, but we’ll fix those in a moment.)

Check for Second Normal form. An entity in Second Normal form is in First Normal, plus all nonkey attributes must depend on the entire primary key. For example, let’s look at a street segment file S (ROUTE <pk>, SEGMENT <pk>, NAME, CONDITION). If ROUTE is defined as “a portion of the street network having the same street name,” then we have a violation of Second Normal form. NAME doesn’t depend on both parts of the primary key, but only on ROUTE. However, CONDITION is in valid Second Normal form.

We would resolve this by decomposing S into two new entities: R (ROUTE <pk>, NAME) and RC (SEGMENT <pk>, CONDITION). We also need an identifying relationship between R and RC, which migrates the ROUTE key, yielding RC (ROUTE <pk, fk>, SEGMENT <pk>, NAME, CONDITION).

Check for Third Normal form. An entity in Third Normal form is in First and Second Normal forms, plus all nonkey attributes must depend on nothing but the primary key. (Sometimes this is combined with the Second Normal form logic and stated as “Attributes depend on the whole key and nothing but the key.”) If data modeling has not clearly distinguished the entities of interest, entities can be commingled.

Third Normal form makes sure that all attributes within an entity are truly descriptive of that entity. Let's look back at our example entities from the First Normal discussion. We got A into First Normal form by decomposing it into B and C, but I indicated that C still had some problems. The problem is that STATUS in A and in C is not solely dependent on the CASE identifier, but also on the DATE. To get C into Third Normal form, we promote DATE to become part of the primary key, yielding C (CASE <pk, fk>, DATE <pk>, STATUS).

(There are two important things to note about what we just did to C. First, we did not have to decompose the entity to solve a normalization problem—we already did that in creating B and C from the original entity A. Second, we also fixed a problem in business logic—the results at the end of resolving the First Normal form problem mandated that a case could have only one status at a time, because the primary key C.CASE must by definition of a primary key be unique.)

Review related entities for impacts of normalization. We've just completed normalization of an entity, which resulted in replacing the original entity A with two new entities and with significant primary key changes. Now, trace out all relationships that A participated in. Restructure these to involve B or C as necessary. Then make any foreign key adjustments to the related entities that are required by this restructuring.

Select another entity and repeat. This process should be performed on every entity in the data model. Remember that you may have to go back to an entity if its structure is modified as a result of the normalization of a related entity.

3) Repeat this overall review cycle until no changes are made. Although we've recommended checking for the impacts of each change along the way, it is possible that a change has not been reviewed. Starting over with this review will ensure that everything has been checked. I seldom need more than three cycles to stabilize a model for a particular set of business rules, and frequently get it to stabilize after one iteration. Of course, review of the model with customers will often find some overlooked or misunderstood business logic that requires changes to the model.

In summary, the normalization process and associated model review gave us several benefits:

- Normalization uncovers entity definition problems. Any problem below the Third Normal form is almost always because of merging two logically distinct real-world things into a single entity.
- Both normalization and the associated structural review of the model will test the internal consistency of the model. Errors of representation of business logic will be caught.
- A well-structured model at a specified level of normalization will prevent certain known problems from occurring during database use.

Normalization does not guarantee a “best” model, but it helps us avoid known bad versions. In combination with reviews with users and other subject matter experts, the normalization process will help ensure the data model is both technically correct and a good representation of the real-world aspects deemed important to our goal.



SECTION 4

WHAT DO YOU DO WITH A DATA MODEL?

The completed data model is used to:

- Create and support a consensus among various groups within the enterprise. There are numerous groups within an enterprise—departments, IT staff, management—that have only a partial understanding of the enterprise derived from their particular perspective. Often these groups start out arguing vociferously about business logic and data requirements. The data modeling process is explicitly designed to generate a consensus on these issues, in part by providing an objective data model from an enterprise perspective to which groups can refer to sort out differences.
- Guide the physical database design and implementation planning. This is discussed briefly in the preceding sections. Obviously, we perform data modeling for a reason (in addition to doing it because it's fun).
- Explain to the DBA what the database is supposed to represent. The technical specialist who must make the database implementation work needs to understand both what he or she is to do (the build-to specifications in the physical design) and also the reasons for that design. Without a clear data model, the DBA may possibly make implementation decisions about such technical issues as performance tuning that actually hinder users in accomplishing their goals through use of the data repository.
- Explain to the programmer what the database is supposed to do. Applications developers build tools to help users interact correctly and efficiently with the data repository. They, too, are guided by the data model in understanding the business logic their tools must support. A substantial amount of the requirements for software engineering is already available in the data model.
- Guide the user to use the database correctly. Users must understand the logic of the data that they are using, especially when they move beyond canned programs and start performing exploratory, ad hoc queries. The data model is a primary source of metadata about the contents of the data repository.



SECTION 5

WHAT'S DIFFERENT ABOUT SPATIAL DATA?

(NOTE: An earlier version of portions of this section was presented in substantially different form at DAMA 2002, “What’s So Spatial about Spatial Data? Integrating Geospatial Data into the Enterprise Data Resource.”)

Those of us within the GIS community tend to assume a geocentric orientation to information technology. We think first of layers or coverages of graphic records and only afterwards about database tables. There is, however, a much broader information technology arena in which people think of data as attributes and seldom if ever consider possible graphical components. Both perspectives tend to see the other perspective as strange and not really very relevant to what they do for a living. In this section, I will argue that both perspectives are wrong—that geospatial data is just one more kind of data to be managed within the enterprise data resource, albeit with some unusual characteristics.

In taking this tack, I need to build some common ground for members of either perspective. I start by discussing how GIS is different.

How GIS Is Different from Other Information Technologies

Geographic Information Systems (GIS) comprise a relatively recent category of information technology. It was developed to work with physical data attributes that use graphical representation in addition to the common digital representation in ASCII or other textual encoding formats. At the lowest level of stored data in computer memory, both types of data are stored in binary format—the differences lie in how the computer “makes sense of” the lists of zeros and ones it encounters.

One important thread in the history of GIS has to do with the progressive differentiation of the graphic data types. Early systems blended spatial and aspatial graphic data elements together. The progressive differentiation of these data types led to separate CAD (computer-aided design) systems for graphic data for which spatial location is not critical from GIS systems for which the (geo)spatial locational characteristics are critical.

In reality, if your data contains location, you already work with geospatial data. And it probably does—remember the old saw that 80 percent of all governmental datasets include some form of location!

The situation is similar in many nongovernmental settings, too. We don't always recognize the data as locational, though, because it usually states location by reference to some other entity—jurisdiction, address, customer premise, facility name, ZIP code, or market region.

In fact, you can do a lot with location data without any graphics. Think of all the mailing-list applications in the world, to take one simple example. These sort and output lists based on ZIP code or state or any number of other spatial location attributes, simply by processing the nongraphic attributes according to instructions (“Give me all customers in ZIP codes 40508 or 40509”).

Where GIS technology really helps is that it makes working with location so much easier. Instead of relying on a ZIP code to select customers for mailing a business promotion flyer or a notice of rezoning request, let's select all parcels within 1,000 feet of the business location!

GIS technologies add these additional functionalities to data processing because:

- They add additional types of data to accommodate the geospatial graphics. Common examples of these data types are polygons (such as jurisdictional boundaries) or points (such as the location of our example business).
- GIS systems add additional types of relationships, derived from the geospatial graphics and their locations relative to one another. In a GIS as contrasted to most CAD systems, each graphic attribute has a set of coordinates associated with it that are tied to specific places on the earth's surface. If the property I own shares coordinates with the area covered by the proposed rezoning, then I get included in the mailing list. (Fortunately, the technical details of the coordinate representation and the calculations that use them are almost completely hidden from

the designer and user of a GIS.) A series of topological relationships can be built on top of this common set of locations, using the mathematics of topology to determine which of the street segments in our centerline file connect to one another and so forth. (Again, the math is hidden from us—we only have to understand how to use the functions provided. Note that this is not so unusual—I may not understand how my relational database handles a statistical function like MODULUS but I can use it!)

To utilize this extra capability, however, we clearly must understand the locational data. We are always better off with metadata, data that documents and describes the contents and structure of the physical database we are using. We frequently take for granted much of this information—once we know we are looking at a date attribute, we just assume it is stored in days expressed in the Common Era calendar and so forth, relying on our common understanding of what a day is. In a GIS, we can get into real trouble “just assuming” all our graphic data is a consistent coordinate system or unit of measure. (Remember the recent Mars probe that crashed because one software team assumed altitude was expressed in meters and another team assumed it was expressed in feet!)

Geographers have spent centuries working out numerous ways to convert a representation of location on the three-dimensional surface of the planet into a symbol on a two-dimensional piece of paper so that certain relationships are preserved. This carries over from the paper map into the computer screen. We need a great deal of additional metadata to understand just how to use the GIS graphic data available to us. Although the end user may take for granted that we designers/implementers got it right for them, someone has to work all this out and document it. (See the Content Standards for Geospatial Metadata promulgated by the U.S. Government's Federal Geospatial Data Committee [FGDC], located at <http://>

www.fgdc.gov/metadata, for a set of standards that outlines all the possible metadata required for the various types of geospatial data. The metadata for nongraphic data is a subset of this total standard.)

A GIS can be defined by examining the components of the name:

- SYSTEM—Integrated set of hardware, software, people, and procedures.
- INFORMATION—Data organized according to business needs and given context for machine or human use.
- GEOGRAPHIC—Having to do with location on the earth's surface.

When we do this, we notice that the only difference from any other information technology system is the geographic element. More technically, we might refer to “geospatial” data, but this debate about what kinds of spatial data might qualify for inclusion under this term is not all that useful to the average practitioner. We can consider a GIS capable in theory of working with locational data on or near the earth's surface from a planetary scale (a world map) or a regional scale (North America) to a site scale (500 W. 3rd ST, Lexington KY USA) and even larger scales (the layout of my home office . . .) Technically, this could extend further, to the molecular level or beyond. Usually, however, we restrict GIS systems to the middle of this range—“earthly phenomena, studied from a spatial perspective at a medium scale (sub-astronomical and super-architectural)” (UCGIS 2006: 45). To further confuse things, there are now GIS implementations based on other planetary systems such as data-mapping systems for Mars exploration.

We have been using “location” as a type of attribute that is related somehow to the graphics that seem to distinguish GIS from other data-management technologies. Remember that a location can be expressed in relative terms (e.g., “within ZIP code 40508”) or in absolute terms (e.g., using latitude/longitude). Now let's look at the manner in which GIS uses locational

attributes to tie the entity instance to a spot on the earth's surface.

The basic principle is that location can be expressed as a set of coordinates, whose data type and domain are defined by the map projection, geoid, and other technical details of the projection from the surface of the globe onto a flat computer screen or map plot. This way of expressing location is represented by “map coordinates,” such as the (possibly) familiar state plane coordinates.

The set of coordinates form a new type of foreign key managed by GIS software, as discussed earlier in the section on relationships. At a relatively low level within the GIS technology, we compare the set of coordinates that describe the location of an instance of an entity to those describing another entity and decide if this quasi-join comparison returns any coordinates that are common to both entity instances. If so, the two participate in a geospatial relationship.

This new capability is how we can talk about “intelligent” maps. If you think of the map as a starting point for database design (and note that the map is better thought of as just one possible type of report), then the design process consists of converting the elements on a map into records in a DBMS, which include both graphic and nongraphic attributes. We can query these records by all the usual DBMS techniques, plus:

- By coordinates
- By pointing
- By overlay and other geospatial operations

This concept of GIS technology as simply an extension to traditional DBMS technologies is difficult for many practitioners to grasp. One major reason is the underlying concept of what a GIS is, which is sometimes expressed as “Big G, Little IS” versus “Little G, Big IS”.

There are two fundamental views of geographic information systems:

G IS—

- Geographic data is unique, different.
- It must be managed differently.
- IT people don't understand it.
- GIS people don't need to understand anything else.

G IS—

- Geographic data is data, with only minor differences from other types.
- It can be managed the same (mostly).
- IT people don't have much trouble understanding it.
- GIS people must understand IT.

In case you can't tell, I fall firmly in the “little G, big IS” camp.

How GIS Integrates with Other Data Management Technologies

Now we are in a position to explore how GIS integrates with other data-management technologies. As I hope the previous section has shown, most aspects of GIS implementation are common to other IT projects. Perhaps the biggest difference is in the new data architecture components necessary to manage the additional data elements—coordinates, graphics, and special metadata.

Over the years, different GIS and CAD vendors have tried different methods of managing the specifically geospatial data. Some vendors have used several methods, either concurrently in products optimized for certain markets or sequentially over time to take advantages of general improvements in IT. (This evolution has also occurred in other data-management technologies such as relational database management systems, but the “black box” nature of these implementations by and large hides the changes in the underlying mechanisms in ways that until recently were not available to GIS vendors.)

Some representative GIS data-management architectures include:

- Flat files—AutoCAD DXF; Intergraph DGN
- Operating system directories—ESRI Arc/Info coverage
- RDBMS table(s)—Oracle Spatial; ESRI SDE
- Objects—Smallworld; ESRI ArcGIS

Specialists may argue over the relative merits of one or another of these and other approaches. For most of us, they all work reasonably well. During physical database design we just have to be cognizant of the peculiarities of the chosen tool, and may have to work harder with one system to implement our logical model than with another system.

The different systems all support the most common types of GIS data (though again with some variations). The most common geospatial vector graphics data types include: Points, Lines (either simple or multipart), and Polygons (again either simple or multipart). These can be assembled into more complex data types such as Networks, which are made up of connected sets of lines. For logical modeling, these provide sufficient data types to capture the business requirements—“We need to show the area of each parcel, so we'll use polygons.” The technical details of how Product X implements a polygon can be left to the physical design, just as we would defer the equivalent details of how it implements a decimal number.

The actual process for specifying the logical data model for geospatial data is straightforward. We start with the fact of existence of the entity, then along with the nongraphic attributes we decide if the business needs to know:

- Absolute location, which tells us if we need some type of graphic attribute, and if so how many dimensions (two, three, or more) will

be required to specify the location.

- Extent or simply location of the entity, which will tell us what type of graphic will be required.
- Geometric shape, which is another way of deciding if simple graphic types will be adequate.
- Topology, which determines what kinds of graphic-based referential integrity must be maintained among elements of the graphic representation, such as connectivity or adjacency.

Adjustments to Database Design Process

Clearly, adding geospatial attributes to an entity includes some additional complexity of attribution. How does this complexity force us to make adjustments to the database design process described in earlier sections?

1) At the conceptual design stage, no adjustments are required. We may find some new conceptual entities that are explicitly geospatial in nature, such as “city limits”.

2) At the logical model stage, we have to add some new questions to our assessment of the attributes:

- Does an entity require locational attributes?
- Can these locations be codings to relative location (e.g., census tract) or must they be absolute coordinates?
- How much detail, or precision, is needed to describe the location?
- Can location be NULL? How would we interpret a missing locational attribute?

3) At the physical design stage, we have to make some significant additions or modifications to the process to work efficiently with the selected implementation technology.

- What are the available vendor-supported storage formats? Does one provide a better fit to the needs of our database? What constraints would a choice place on our implementation planning?
- What degree of integration is possible between graphic and nongraphic attributes? Between the GIS storage and the external systems? (It is preferable to keep the data-management architecture as simple as possible.)
- What coordinate system, projection, etc., will the graphics be expressed in? Is the required projection, such as a state-specified projection for regional mapping, supported by the selected product?
- What is the nominal resolution and accuracy of measurement that is required? Environmental or urban planning users may be satisfied with data of +/-20 feet, but not the engineers or surveyors!
- What feature of geometry (e.g., simple or multipart polygons) will we use?
- What types of spatial indexing are available to speed operations upon coordinates?

4) At the physical implementation stage, the distinct characteristics of the chosen implementation technology really start to affect the decision-making process. For the traditionally trained DBAs, these can be especially challenging. A list of the more common challenges includes:

- DBA toolkit to use
- Unfamiliar DBMS data types and indexes
- New and complex storage calculations
- Nonstandard table structures from GIS “Middleware” toolkits
- Nonstandard backup/restore
- Very, very long transactions
- Dual tuning modes—graphic and nongraphic

In Conclusion

1. GIS is like any other data-management technology.
2. GIS also involves understanding cartographic data and topologic concepts.
3. GIS implementation forces changes to physical design and implementation.



SECTION 6

WHATS DIFFERENT ABOUT OBJECT MODELING?

In the past decade, object-oriented (OO) approaches to data modeling have moved into the mainstream of data resource management. This move involves increasing integration of OO and other data-management architectures within enterprise systems, whether through hybrid object-relational DBMSs or object approaches to GIS data. However, the either/or mentality that characterized early debates over OO programming still persists in the modeling community.

In this section, I will first summarize the basics of object-oriented approaches to data resource management. Then I will review OO methods of design. I will close with an assessment of how OO technologies change the approach to data modeling presented in this document. Note that this is by no means a tutorial on the Unified Modeling Language—see Fowler and Scott 1999, Cockburn 1998, and Eriksson and Penker 1998 (among many, many others) for an effective introduction to UML and its usage.

Overview of Object Orientation

Object-orientation (OO) began as an approach to programming, where it emphasized the use of coherent “objects” that bundle identity, attribution,

and behavior into a unit. This was felt to increase the intelligence of the object, for (a) these highly inter-related aspects of the object were bundled together and encapsulated into a unit and (b) these aspects were consistently and completely implemented whenever the program creates an instance of the specified object class.

Object programming quickly excelled in the growing category of GUI-based applications, including those using the Internet. Unlike a command-line application, a GUI-based application may allow the user multiple possible actions at any moment. Think of how many possible actions are available to me at this moment when I am writing this sentence using a widely used word processor. Object instances have the advantage of being responsive in predictable ways to messages from other object instances, which makes them perfect for message-oriented applications. Furthermore, they persist in memory until removed and maintain their state between interactions. Many other application categories took over this behavior to improve the user interaction with the application, including most GIS systems.

These early applications of OO institutionalized what I find to be the biggest obstacle to conducting data

modeling within an OO context. OO has evolved to better fit object instances implemented in memory via an application, rather than persistent data stores. To clarify the dual nature of most objects in a GIS implementation: The object class defines the behavior and internal, transient data structures that are created at the time of instantiation, but the class definition may also include a link to a persistent data store so the facts stored in the object instance during its lifetime can be preserved and reused later.

This growth brought object-oriented methods into the forefront of computing. Their obvious strengths went hand in hand with the needs of new approaches to applications, strengthening both. This had two unfortunate side effects, however. First, the brashness typical of proponents of upstart methods that was displayed by OO advocates seeking to win legitimacy can all too easily become dogmatic once that legitimacy was won. Another concern is that OO methods work best in event-driven applications, which leads to a relative deprecation of persistence between events.

OO design methods, therefore, tend to emphasize the behavior of objects in response to events, and relatively neglect the knowledge the objects contain. The current state of the art in OO design is formalized in the Unified Modeling Language (UML) that is managed by the Object Management Group (OMG). OMG is currently upgrading all the UML to Version 2.0 from Version 1.5, the previous release. “Adoption of the UML 2.0 Superstructure is complete—No further technical work is being done; in fact the Superstructure specification has been stable since it took its adopted form in October, 2004. The superstructure defines the six structure diagrams, three behavior diagrams, four interaction diagrams, and the elements that comprise them. . . .” (<http://www.uml.org>; last accessed July 21, 2006).

UML, however, does not have a formal notation for persistent data storage and management. Until very recently, there are two basic approaches to integrating database design into UML: either one extends

the standard notation through mechanisms such as stereotypes or one uses a hybrid object-and-relational mix of techniques.

Neither of these approaches is as easy as it might seem. Relational database and OO technologies were developed independently of one another, using profoundly different ways of thinking about the world and about software systems. This leads to significant translation barriers—one cannot simply assume that you can convert an object class into a database table or vice versa. The term for this in the OO literature is impedance mismatch—see Ambler (2003:105-113) for an extended discussion and the rest of the book for solutions. Ambler notes that the impedances work on several levels. First, there is a cultural impedance when practitioners of OO design and of database design tend to work in incompatible ways, which is part of his rationale for bringing agile methods to bear as a means to minimize problems on teams with members from both camps. (Cultural impedance mismatch is also the primary theme of this section.) Additionally, there are technological impedance mismatch issues around such issues as indexing in the database or refactoring object classes that can trip the unwary.

Recently, people have begun formalizing methodologies for OO database design (Naiburg and Maksimcuk 2001, Harrington 2000). These usually build upon UML by adding new notations or by using standard notation elements in innovative ways. This continues an important trend within OO methodology—UML does not contain a methodology, only a consistent set of notations for utilization within a methodology of choice (Rosenberg and Scott 1999).

Database Design for Object Orientation

It is important to recognize that the early OO methodologies tended to downplay if not ignore the persistent data-storage issue, at best considering the object class model as a substitute for database design.

Often this was justified because the OO project was to develop an application that could take existing databases as static and “given” within the application environment.

As OO methods began to be applied across a wider range of IT applications, the limitations of this approach became increasingly evident. Several approaches to designing databases within an object-oriented process have recently been proposed.

- Use OO tools to manage persistence within an object framework, through either true OO databases (such as Objectivity/DB or ObjectStore) or through object extensions to relational databases.
- Use OO design methods, then transform the class definitions into structures within a relational database.
- Extend OO design methods through stereotypes to the class diagram, for modeling relational database elements. In UML notation, a stereotype is a defined mechanism “to classify elements so that they behave in ways the UML doesn’t formally define.” (Rosenberg and Scott 1999:62)
- Design the relational database using traditional methods, then transform the definitions of tables and other relational structures into objects.

The Uniform Modeling Language (UML) is the most widely used set of diagrams and descriptive notations within the object-oriented community. “UML 2.0 defines 13 types of diagrams, divided into 3 categories: 6 diagram types represent the static application structure; 3 represent general types of behavior; and 4 represent different aspects of interactions:

- Structure Diagrams include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

- Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering), the Activity Diagram, and the State Machine Diagram.
- Interaction Diagrams, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.” (OMG, http://www.omg.org/gettingstarted/what_is_uml.htm; last visited July 21, 2006)

UML and the growing industry of UML explication also provide standards and best practices for their utilization, though not a methodology to guide their application. Several diagrams within UML, especially the Use Case during conceptual and logical modeling and the Sequence Diagram during physical design, can be highly useful and effective means of defining the scope and context of the data resource, but none of the currently recognized UML diagram types are substitutes for Entity-Relationship Diagrams and related tools in the data modeling toolkit.

The Use Case is the most widely adopted component of UML. I find Use Cases of limited use for general-purpose data resource design. They can be great for gathering requirements and identifying the need for persistence in an OO application, but may need additional focus to document how the data is to be handled. I would also stress the importance of the Use Case document rather than the diagram. Some people think the diagram is something more than a graphic index into the real-use cases.

The Sequence Diagram (aka “Swimlane” Diagram) is an effective means of describing the physical implementation details of the data flow. The concept of a message passed from class instance to class instance includes the data elements being passed. When one of the classes is a database table, the dynamics of the data resource can be cleanly and completely specified.

Within UML, the most effective diagram type for data modeling is the Class Diagram. (Note: Some CASE tools and other implementations of UML distinguish between generic Class Diagrams and static Class Diagrams. The difference is primarily the ability to stipulate persistence implicitly. “Static” in UML is the opposite of “dynamic”; it ignores changes over time. Other implementations use only a single type of Class Diagram, but support varying behavior through a set of abstract classes and/or stereotypes that they provide.) The Class Diagram, like the ERD, can be used at different levels of detail to correspond to the conceptual, logical, and physical design stages. Furthermore, because a class defines both the data elements and the behavior that comprise it, one can emphasize different aspects of design to meet the needs of the project.

Impacts of Object Orientation on the Approach in This Book

I believe that object-orientated approaches have real benefits, which can help us adapt to changing demands for data modeling. Changes in tools and techniques on the part of technology vendors are forcing us to move into increasingly sophisticated and demanding database designs.

However, I am not convinced that OO approaches are best practices for all phases of database design. The approach has real benefit if the physical implementation is constrained to use an OO or quasi-OO toolkit, but even then I think there are much better ways of handling conceptual and logical design. So, I would summarize my position as recognizing OO methods as one among several good practices, but not necessarily always a best practice (even if you have committed to explicitly OO data-management tools).

The approach to various stages of the database design life cycle presented in this book is chosen for maximum adaptability to fit the business requirements and the implementation toolkit. Within a large

enterprise there are almost certainly a broad range of data-management tools, from legacy mainframe systems using flat files or hierarchical databases to object-oriented GIS datasets. The challenge to IT professionals is to effectively integrate this disparate set of data resources to maximize benefits to the enterprise.

The life-cycle approach described here, based on IDEF1X, supports this integration goal by rigid separation between the logical and physical stages. The conceptual and logical stages are implementation-agnostic; they focus on business requirements without worrying about how to implement the entities being modeled.

The conceptual design stage gives two critical benefits supporting this goal: (a) You find out whether the database is intended to be part of a project-specific or application-specific exercise, or to be a general-purpose, enterprise repository; and (b) You get grounds for deciding what aspects of the real-world things to be represented are relevant. These clarifications facilitate proper deployment of OO methods because OO in general is much better at project-specific applications than it is at designing persistent, multipurpose data resources.

The implementation neutrality of the logical modeling stage is essential for this process. I strongly believe the logical model must focus on documenting the enterprise rules and meanings, independent of the implementation(s) chosen to make real systems that correspond to the logical model. This is the only way I know to be sure you can, for example, link a textual database at the Assessor’s office to the building object class in the GIS, through clarifying the commonalities and differences in definition between the Assessor’s “improvement” and the GIS “building”.

Once you have this firm foundation of a logical model, the generation of a physical database design for an OO implementation technology can proceed much as it would for a relational implementation. Because each physical implementation mandates dif-

ferent terminologies and techniques to accommodate the unique characteristics of the implementation tool, OO tools just add one more set of distinct characteristics to be mastered by the designer.

In summary, I see two critical potential errors in OO database design. First, designers can lose the critical distinction among stages of design, with implementation specifics reserved for physical design and implementation planning. (This is hardly unique to OO

advocates; it is a constant temptation for all designers, especially those from a programming background.) Second, designers (or their colleagues) falsely believe that OO database design is fundamentally different than other approaches, and that all the painstaking logical modeling can be skipped.

I hope this section and the argument throughout this book convince you otherwise.



APPENDIX

— RESOURCES FOR THE DATA MODELER

The following lists of resources are not intended to provide complete inventories, nor should they be taken as recommendations or endorsements by URISA or the author. The only intent is to provide a starting point for intelligent shopping for those seeking more information or tools for performing data modeling.

The items in the following list were selected because the author knew about them or discovered them in the course of developing this guide, and has found them useful either in their own right or as having links to additional resources that might prove useful. Many other titles could be added to this bibliography; the Data Management Association (DAMA) Web site has a multipage bibliography that will prove useful to the reader seeking further information, especially on related topics not covered in this study.

ORGANIZATIONS

Data Management Association (DAMA) <http://www.dama.org>

This is the premier technical organization for those involved in data modeling or database design. It also contains numerous links to jour-

nals, other organizations, practitioners' personal and/or corporate sites, and to local chapters.

Object Management Group (OMG) <http://www.omg.org>; <http://www.uml.org>

The OMG is the nonprofit organization for the specification and dissemination of standards and methods for working with objects. One of its more successful standards efforts is the Unified Modeling Language (UML).

Urban and Regional Information Systems Association (URISA) <http://www.urisa.org>

URISA is a broad-based organization involving all aspects of information technology as practiced within state and local government. It has a strong concentration on GIS as well as database management, and provides services to both the technical practitioner and the administrator.

Zachman Institute for Framework Advancement (ZIFA). <http://www.zifa.org>.

The mission of the Zachman Institute for Framework Advancement (ZIFA) is “to exercise the Zachman Framework for Enterprise Architecture, for the purpose of advancing the conceptual and implementation understanding of Enterprise Architecture.”

PUBLICATIONS

Ambler, Scott W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York: Wiley, 2003.

As a follow-up to his excellent *Agile Modeling*, the author applies his agile methods specifically to data-centric projects. He notes that agile methods can benefit the data resource management specialists as well as the developers on a project team, and demonstrates the point for both collaborating on an agile project and for traditional data-management functions. Especially useful are Chapter 14, “Mapping Objects to Relational Databases,” and Chapter 18, “Finding Objects in Relational Databases.”

ANSI/X#/SPARC Study Group on Data Base Management Systems. Interim Report. *ACM SIGMOD Bulletin* 7(2), 1975.

The most frequently cited report on this study group’s findings. The three-schema architecture has been discussed and referenced in numerous studies since this publication.

Atre, S. *Data Base: Structured Techniques for Design, Performance, and Management, with Case Studies*. New York: John Wiley & Sons, A Wiley-Interscience Publication, 1980.

A presentation of structured methods as applied to data modeling. It covers a broader range of topics than does this study.

Brackett, Michael H. *Practical Data Design*. Englewood Cliffs, NJ: Prentice Hall, 1990.

Brackett presents a wide-ranging discussion of data design and implementation issues, organized according to his business-oriented approach. The user will find this a useful reference, referring as needed to chapters on normalization, naming conventions, and similar useful topics.

Bruce, Thomas A. *Designing Quality Databases with IDEF1X Information Models*. New York: Dorset House Publishing, 1992.

An excellent handbook for data modelers, of broader benefit than the title might imply. This is the best training guide I’ve seen for IDEF1X implementation. However, the author is very good at generalizing his presentation so it can be of benefit to those who choose another methodology.

Byte. “Methodology: The Experts Speak.” *Byte* 14(4): 221–233, 1989. Contains short articles from five leading theorists in systems design: Ken Orr, “The Warnier/Orr Approach”; Chris Gane, “The Gane/Sarson Approach”; Edward Yourdon, “The Yourdon Approach”; Peter Chen, “The Entity-Relationship Approach”; and Larry L. Constantine, “The Structured Design Approach.”

If you can find this, it provides a brief but insightful presentation of these approaches to systems design. The short presentations by the theorists both help indicate how they summarize their work, and in several instances indicate their opinions of how their work compares to other approaches.

Chen, Peter. *The Entity-Relationship Approach to Logical Data Base Design*. The Q.E.D. Monograph Series, Data Base Management Number 6. Wellesley, MA: Q.E.D. Informational Sciences, Inc., 1977.

A complete presentation of Dr. Chen's methodology, which updates the initial presentation with the benefits of several years of additional experience. Many theorists have claimed to extend or replace this methodology, but the tool of Entity-Relationship Diagramming is still the most commonly used tool in data modeling.

Cockburn, Alistair. *Surviving Object-Oriented Projects: A Manager's Guide*. Booch, Jacobson, Rumbaugh Object Technology Series. Boston: Addison-Wesley, 1998.

Focuses on the technical issues relative to setting up and executing an object-oriented project. Useful cautionary material for those who are excessively sanguine about OO as a "silver bullet" for all IT woes; also useful resource for those committed to success (and survival in a major IT project is certainly one measure of success!).

Date, C. J. *An Introduction to Database Systems*. Sixth Edition. Reading, MA: Addison-Wesley Publishing, 1995.

Date is one of the more prolific and readable theorists in the field of database systems. This widely read volume, which receives significant rewrites between most editions, is his attempt at synthesis of his understanding of database design and implementation. Date's sometimes controversial opinions tend to focus more on using a database than on designing one, but the two perspectives are obviously closely connected. This is currently in its Eighth Edition, but I've cited the one I'm familiar with.

Dutka, Alan F., and Howard H. Hanson. *Fundamentals of Data Normalization*. Reading, MA: Addison-Wesley, 1989.

A brief technical presentation of data normalization, using the mathematics of sets. It includes

examples and discussions that are worthwhile to the more nontechnical user.

Eriksson, Hans-Erik, and Magnus Penker. *UML Toolkit*. New York: John Wiley & Sons, 1998.

A one-volume orientation to the Unified Modeling Language, which can serve as a starting point before getting into the more technical presentations now becoming available.

Federal Geographic Data Committee, U.S. Government. (1998) Content Standard for Geospatial Metadata, Version 2. (FGDC-STD-001-1998) http://www.fgdc.gov/standards/projects/FGDC-standards-projects/metadata/base-metadata/index_html; last accessed July 24, 2006.

Provides a well-worked-out standard for geospatial metadata, which is a superset of the general attribute-only metadata presented as Section 5 of the standard. (If you are unfamiliar with this standard, don't worry about the volume of metadata elements—most are optional and relevant to only a small subset of all possible geospatial data. For instance, if you don't manage aerial photography, you have no reason to care about altitude, focal length, and all the other metadata elements for photography.)

Finkelstein, Clive. *Information Engineering: Strategic Systems Development*. Sydney, Australia: Addison-Wesley, 1992.

Finkelstein has integrated his version of Information Engineering with strategic planning methodologies to produce a methodology strongly focused on business logic. For data modelers, the sections on solving frequently encountered data structures and the discussion on Business Normalization are especially worthwhile.

Fowler, Martin, and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Second Edition. Booch, Jacobson, Rumbaugh Object Technology Series. Reading, MA: Addison-Wesley, 1999.

The best single-volume discussion of UML notation. Has limited discussion of data modeling per se, but indispensable for users of UML.

Harrington, Jan L. *Object Oriented Database Design Clearly Explained*. San Francisco: Morgan-Kaufmann, 2000.

A parallel text to *Relational Database Design Clearly Explained*, this book attempts to transfer relationally based design practices to OO database design.

Hay, David C. *Data Model Patterns: Conventions of Thought*. New York: Dorset House, 1996.

Applies the Design Patterns approach to data modeling. Makes a good case for approaching data modeling through emphasis on standardized, generalized patterns. This approach is based on the assumption that “the underlying structures of enterprises are similar, or at least that they have similar components” (Preface, p. xix). Hay also makes the point that design patterns are appropriately used to generate a starting model that can be customized to fit the peculiarities of the specific situation, and not necessarily the final model for implementation.

Hernandez, Michael J. *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design*. Reading, MA: Addison-Wesley Developers Press, 1997.

Provides a solid nontechnical introduction to data modeling.

Martin, James. *Information Engineering. Book II: Planning and Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1990.

Part two of a trilogy on Information Engineering as James Martin has evolved the methodology. Provides a solid discussion of data modeling from the enterprise perspective, but as one part of a much more global study.

Naiburg, Eric J., and Robert A. Maksimcuk. *UML for Database Design*. Booch, Jacobson, Rumbaugh Object Technology Series. Boston: Addison-Wesley, 2001.

The authors tout UML as a common language to unify the different IT specialists involved in developing and supporting complex, enterprise systems. They see this as a highly desirable antidote to excessively data-centric approaches to enterprise IT. Part of their justification is that many aspects of the enterprise operations cannot be captured within the database. Especially useful is Chapter 6, “Preparing for Transformation to the Database Design Model.”

National Institute of Standards and Technology (NIST). *Integration Definition for Information Modeling (IDEF1X)*. Federal Information Processing Standards Publication 184. Washington, D.C.: U.S. Department of Commerce, National Institute of Standards and Technology, December 21, 1993.

The technical specification of the Federal Information Processing Standard for IDEF1X. It contains a useful historical context for the evolution of the methodology, a succinct specification, and some helpful explication of terms.

Orr, Kenneth T. *Structured Systems Development*. NY: Yourdon Press, 1977.

One of the essential books on structured methods of software development, and perhaps the most useful for data modelers.

Rational Software Corporation. *The UML and Data Modeling*. Rational Whitepaper TP-180 3/00. Cupertino, CA: Rational Software Corporation, 2000.

Presents an approach to using Class Diagrams for data modeling, via a proposed stereotype.

Reingruber, Michael C., and William W. Gregory. *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*. Hoboken, NJ: John Wiley & Sons, Inc., 1994.

The approach to defining and ensuring the quality of a data model is useful, even if one might quibble about the authors' criteria for quality. Much of the material is presented in a problem-resolution mode that makes this an excellent reference.

Rosenberg, Doug, and Kendall Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Booch, Jacobson, Rumbaugh Object Technology Series. Boston: Addison-Wesley, 1999.

A useful one-volume distillation of the UML notation, coupled with a recommended, proprietary methodology for utilization. The authors' approach to object modeling does not extensively deal with data modeling concerns, but their approach to domain modeling is especially useful for linking object-modeling processes to conceptual/logical stages of the data modeling process presented in this study.

Saake, G., S. Conrad, I. Schmitt, and C. Turker. *Object-Oriented Database Design: What Is the Difference with Relational Database Design*. In *ObjectWorld Frankfurt '95—Conference Notes, Software Development Track SD.6*, pp. 1–10, October, 1995.

Summarizes the technical and theoretical benefits of using an object-oriented design process, though I find the comparison with relational modeling approaches somewhat one-sided.

Spewak, Steven H., and Steven C. Hill. *Enterprise Architecture Planning: Developing a Blueprint for Data, Applications, and Technology*. Boston: QED Publishing, 1992.

Provides a variant methodology for enterprise planning, based heavily on Information Engineering concepts. The volume is structured in a work breakdown structure for performing an in-house enterprise architecture planning study, including the enterprise data architecture.

Ullman, Jeffrey D. *Principles of Database and Knowledge-Base Systems*. Volume I. *Principles of Computer Science Series*, Number 14. Stanford, CA: Computer Science Press, 1988.

A classic presentation of database systems from a computer-science perspective. It will repay a reading, even for those of us lacking the CS background Ullman presupposes.

University Consortium for Geographic Information Science (UCGIS). *Geographic Information Science and Technology Body of Knowledge 2006*. <http://www.ucgis.org/priorities/education/modelcurriculumproject.asp#1>; last accessed April 26, 2006.

Provides an outline of the “body of knowledge” for GI S&T, in a format compatible with those for

other disciplines such as project management.
This includes many topics relevant to this study.

Zachmann, John. A Framework for Information Systems Architecture. IBM Systems Journal 26(3), 1987. IBM Publication G321-5298.

The seminal article introducing the Zachmann Framework for Enterprise Architecture. See extensive bibliography and some recent papers by Zachman at the Web site of the Zachman Institute for Framework Advancement (<http://www.zifa.org>).