CHAPTER
2

The Dangers in a Relational Database

As with any new technology or new venture, it's sensible to think through not only the benefits and opportunities that are presented, but also the costs and risks. Combine a relational database with a series of powerful and easy-to-use tools, as Oracle does, and the possibility of being seduced into disaster by its simplicity becomes real. Add in object-oriented and web capabilities, and the dangers increase. This chapter discusses some of the dangers that both developers and users need to consider.

Is It Really as Easy as They Say?

According to the database vendors—the industry evangelists—developing an application using a relational database and the associated "fourth-generation" tools will be as much as 20 times faster than traditional system development. And it will be very easy: ultimately, programmers and systems analysts will be used less, and end users will control their own destinies.

Critics of the relational approach warn that relational systems are inherently slower than others, that users who are given control of query and report writing will overwhelm computers, and that a company will lose face and fortune if a more traditional approach is not taken. The press cites stories of huge applications that simply failed to run when they were put into production.

So, what's the truth? The truth is that the rules of the game have changed. Fourth-generation development efforts make very different demands upon companies and management than do more traditional methods. There are issues and risks that are brand new and not obvious. Once these are identified and understood, the risk is no greater, and probably much smaller, than in traditional development.

What Are the Risks?

The primary risk is that developing relational database applications *is* as easy as they say. Understanding tables, columns, and rows isn't difficult. The relationship between two tables is conceptually simple. Even *normalization,* the process of analyzing the inherent or "normal" relationships between the various elements of a company's data, is fairly easy to learn.

Unfortunately, this often produces instant "experts," full of confidence but with little experience in building real, production-quality applications. For a tiny marketing database, or a home inventory application, this doesn't matter very much. The mistakes made will reveal themselves in time, the lessons will be learned, and the errors will be avoided the next time around. In an important application, however, this is a sure formula for disaster. This lack of experience is usually behind the press's stories of major project failures.

Older development methods are generally slower, primarily because the tasks of the older methods—coding, submitting a job for compilation, linking, and testing—result in a slower pace. The cycle, particularly on a mainframe, is often so tedious that programmers spend a good deal of time "desk-checking" in order to avoid going through the delay of another full cycle because of an error in the code.

Fourth-generation tools seduce developers into rushing into production. Changes can be made and implemented so quickly that testing is given short shrift. The elimination of virtually all desk-checking compounds the problem. When the negative incentive (the long cycle) that encouraged desk-checking disappeared, desk-checking went with it. The attitude of many seems to be, "If the application isn't quite right, we can fix it quickly. If the data gets corrupted, we can patch it with a quick update. If it's not fast enough, we can tune it on the fly. Let's get it in ahead of schedule and show the stuff we're made of."

This problem is made worse by an interesting sociological phenomenon: Many of the developers of relational applications are recent college graduates. They've learned relational or object-oriented theory and design in school and are ready to make their mark. More seasoned developers, as a class, haven't learned the new technology. They're busy supporting and enhancing the technologies they know, which support their companies' current information systems. The result is that inexperienced developers tend to end up on the relational projects, are sometimes less inclined to test, and are less sensitive to the consequences of failure than those who have already lived through several complete application development cycles.

The testing cycle in an important Oracle project should be longer and more thorough than in a traditional project. This is true even if proper project controls are in place, and even if seasoned project managers are guiding the project, because there will be less desk-checking and an inherent overconfidence. This testing must check the correctness of data entry screens and reports, of data loads and updates, of data integrity and concurrence, and particularly of transaction and storage volumes during peak loads.

Because it really is as easy as they say, application development with Oracle's tools can be breathtakingly rapid. But this automatically reduces the amount of testing done as a normal part of development, and the planned testing and quality assurance must be consciously lengthened to compensate. This is not usually foreseen by those new to either Oracle or fourth-generation tools, but you must budget for it in your project plan.

The Importance of the New Vision

Many of us look forward to the day when we can simply type a "natural" language query in English, and have the answer back, on our screen, in seconds.

We are closer to this goal than most of us realize. The limiting factor is no longer technology, but rather the rigor of thought in our application designs. Oracle can straightforwardly build English-based systems that are easily understood and exploited by unsophisticated users. The potential is there, already available in Oracle's database and tools, but only a few have understood and used it.

Clarity and understandability should be the hallmarks of any Oracle application. Applications can operate in English, be understood readily by end users who have no programming background, and provide information based on a simple English query.

How? First of all, a major goal of the design effort must be to make the application easy to understand and simple to use. If you err, it must always be in this direction, even if it means consuming more CPU or disk space. The limitation of this approach is that you could make an application exceptionally easy to use by creating overly complex programs that are nearly impossible to maintain or enhance. This would be an equally bad mistake. However, all things being equal, an end-user orientation should never be sacrificed for clever coding.

Changing Environments

Consider that the cost to run a computer, expressed as the cost per million instructions per second (MIPS), has historically declined at the rate of 20 percent per year. Labor costs, on the other hand, have risen steadily, not just because of the general trend, but also because salaries of individual employees increase the longer they stay with a company and the better they become at their jobs. This means that any work that can be shifted from human laborers to machines is a good investment.

Have we factored this incredible shift into our application designs? The answer is "somewhat," but terribly unevenly. The real progress has been in *environments,* such as the visionary work first done at Xerox Palo Alto Research Center (PARC), and then on the Macintosh, and now in MS-Windows, web-based browsers, and other graphical, icon-based systems. These environments are much easier to learn and understand than the older, character-based environments, and people who use them can produce in minutes what previously took days. The improvement in some cases has been so huge that we've entirely lost sight of how hard some tasks used to be.

Unfortunately, this concept of an accommodating and friendly environment hasn't been grasped by many application developers. Even when they work in these environments, they continue old habits that are just no longer appropriate.

Codes, Abbreviations, and Naming Standards

The problem of old programming habits is most pronounced in codes, abbreviations, and naming standards, which are almost completely ignored when the needs of end users are considered. When these three issues are thought about at all, usually only the needs and conventions of the systems groups are considered. This may seem like a dry and uninteresting problem to be forced to think through, but it can make the difference between great success and grudging acceptance, between an order-of-magnitude leap in productivity and a marginal gain, between interested, effective users and bored, harried users who make continual demands on the developers.

Here's what happened. Business records used to be kept in ledgers and journals. Each event or transaction was written down, line by line, in English. As we developed applications, codes were added to replace data values (such as "01" for "Accounts Receivable," "02" for "Accounts Payable," and so on). Key-entry clerks would actually have to know or look up most of these codes and type them in at the appropriately labeled fields on their screens. This is an extreme example, but literally thousands of applications take exactly this approach and are every bit as difficult to learn or understand.

This problem has been most pronounced in large, conventional mainframe systems development. As relational databases are introduced into these groups, they are used simply as replacements for older input/output methods such as Virtual Storage Access Method (VSAM) and Information Management System (IMS). The power and features of the relational database are virtually wasted when used in such a fashion.

Why Are Codes Used Instead of English?

Why use codes at all? Two primary justifications are usually offered:

- A category has so many items in it that all of them can't reasonably be represented or remembered in English.
- To save space in the computer.

The second point is an anachronism. Memory and permanent storage were once so expensive and CPUs so slow (with less power than a modern hand-held calculator) that programmers had to cram every piece of information into the smallest possible space. Numbers, character for character, take half of the computer storage space of letters, and codes reduce the demands on the machine even more.

Because machines were expensive, developers had to use codes for *everything* to make *anything* work at all. It was a technical solution to an economic problem. For users, who had to learn all sorts of meaningless codes, the demands were terrible. Machines were too slow and too expensive to accommodate the humans, so the humans were trained to accommodate the machines. It was a necessary evil.

This economic justification for codes vanished years ago. Computers are now fast enough and cheap enough to accommodate the way people work, and use words that people understand. It's high time that they did so. Yet, without really thinking through the justifications, developers and designers continue to use codes.

The first point—that of too many items per category—is more substantive, but much less so than it first appears. One idea is that it takes less effort (and is therefore less expensive) for someone to key in the numeric codes than actual text string values like book titles. This justification is untrue in Oracle. Not only is it more costly to train people to know the correct customer, product, transaction, and other codes, and more expensive because of the cost of mistakes (which are high with code-based systems), but using codes also means not using Oracle fully; Oracle is able to take the first few characters of a title and fill in the rest of the name itself. It can do the same thing with product names, transactions (a "b" will automatically fill in with "buy," an "s" with "sell"), and so on, throughout an application. It does this with very robust pattern-matching abilities.

The Benefit of User Feedback

There is an immediate additional benefit: Key-entry errors drop almost to zero because the users get immediate feedback, in English, of the business information they're entering. Digits don't get transposed; codes don't get remembered incorrectly; and, in financial applications, money rarely is lost in accounts due to entry errors, with significant savings.

Applications also become much more comprehensible. Screens and reports are transformed from arcane arrays of numbers and codes into a readable and understandable format. The change of application design from code-oriented to English-oriented has a profound and invigorating effect on a company and its employees. For users who have been burdened by code manuals, an English-based application produces a tremendous psychological release.

How to Reduce the Confusion

Another version of the "too many items per category" justification is that the number of products, customers, or transaction types is just too great to differentiate each by name, or there are too many items in a category that are identical or very similar (customers named "John Smith," for instance). A category can contain too many entries to make the options easy to remember or differentiate, but more often this is evidence of an incomplete job of categorizing information: Too many dissimilar things are crammed into too broad a category. Developing an application with a strong English-based (or French, German, Spanish, and so on) orientation, as opposed to code-based, requires time spent with users and developers—taking apart the information about the business, understanding its natural relationships and categories, and then carefully constructing a database and naming scheme that simply and accurately reflects these discoveries.

There are three basic steps to doing this:

1. Normalize the data.
2. Choose English names for the tables and columns.
3. Choose English words for the data.

Each of these steps will be explained in order. The goal is to design an application in which the data is sensibly organized, is stored in tables and columns whose names are familiar to the user, and is described in familiar terms, not codes.

Normalization

Relations between countries, or between departments in a company, or between users and developers, are usually the product of particular historical circumstances, which may define current relations even though the circumstances have long since passed. The result of this can be abnormal relations, or, in current parlance, dysfunctional relations. History and circumstance often have the same effect on data—on how it is collected, organized, and reported. And data, too, can become abnormal and dysfunctional.

Normalization is the process of putting things right, making them normal. The origin of the term is the Latin word *norma,* which was a carpenter's square that was used for ensuring a right angle. In geometry, when a line is at a right angle to another line, it is said to be "normal" to it. In a relational database, the term also has a specific mathematical meaning having to do with separating elements of data (such as names, addresses, or skills) into *affinity groups,* and defining the normal, or "right," relationships between them.

The basic concepts of normalization are being introduced here so that users can contribute to the design of an application they will be using, or better understand one that's already been built. It would be a mistake, however, to think that this process is really only applicable to designing a database or a computer application. Normalization results in deep insights into the information used in a business and how the various elements of that information are related to each other. This will prove educational in areas apart from databases and computers.

The Logical Model

An early step in the analysis process is the building of a *logical model,* which is simply a normalized diagram of the data used by the business. Knowing why and how the data gets broken apart and segregated is essential to understanding the model, and the model is essential to building an application that will support the business for a long time, without requiring extraordinary support.

Normalization is usually discussed in terms of *form*: First, Second, and Third Normal Form are the most common, with Third representing the most highly normalized state. There are Fourth and Fifth normalization levels defined as well, but they are beyond the scope of this discussion.

Consider a bookshelf; for each book, you can store information about it—the title, publisher, authors, and multiple categories or descriptive terms for the book. Assume that this book-level data became the table design in Oracle. The table might be called BOOKSHELF, and the columns might be Title, Publisher, Author1, Author2, Author3, and Category1, Category2, and Category3.

The users of this table already have a problem: in the BOOKSHELF table, users are limited to listing just three authors or categories for a single book.

What happens when the list of acceptable categories changes? Someone has to go through every row in the BOOKSHELF table and correct all the old values. And what if one of the authors changes his or her name? Again, all of the related records must be changed. What will you do when a fourth author contributes to a book?

These are not really computer or technical issues, even though they became apparent because you were designing a database. They are much more basic issues of how to sensibly and logically organize the information of a business. They are the issues that normalization addresses. This is done with a step-by-step reorganization of the elements of the data into affinity groups, by eliminating dysfunctional relationships, and by ensuring normal relationships.

Normalizing the Data Step one of the reorganization is to put the data into First Normal Form. This is done by moving data into separate tables, where the data in each table is of a similar type, and giving each table a *primary key*—a unique label or identifier. This eliminates repeating groups of data, such as the authors on the bookshelf.

Instead of having only three authors allowed per book, each author's data is placed in a separate table, with a row per name and description. This eliminates the need for a variable number of authors in the BOOKSHELF table and is a better design than limiting the BOOKSHELF table to just three authors.

Next, you define the primary key to each table: What will uniquely identify and allow you to extract one row of information? For simplicity's sake, assume the titles and authors' names are unique, so AuthorName is the primary key to the AUTHOR table.

You now have split BOOKSHELF into two tables: AUTHOR, with columns AuthorName (the primary key) and Comments, and BOOKSHELF, with a primary key of Title, and with columns Publisher, and Category1, Category2, and Category3, Rating, and RatingDescription. A third table, BOOKSHELF_AUTHOR, provides the associations: Multiple authors can be listed for a single book and an author can write multiple books—known as a many-to-many relationship. Figure 2-1 shows these relationships and primary keys.

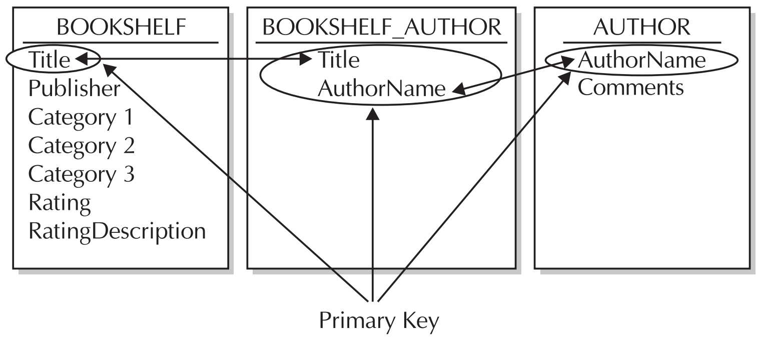

FIGURE 2-1. *The BOOKSHELF, AUTHOR, and BOOKSHELF_AUTHOR tables*

The next step in the normalization process, Second Normal Form, entails taking out data that's only dependent on a part of the key. If there are attributes that do not depend on the entire key, then those attributes should be moved to a new table. In this case, RatingDescription is not really dependent on Title—it's based on the Rating column value, so it should be moved to a separate table.

The final step, Third Normal Form, means getting rid of anything in the tables that doesn't depend solely on the primary key. In this example, the categories are interrelated; you would not list a title as both "Fiction" and "Nonfiction," and you would have different subcategories under the "Adult" category than you would have under the "Children" category. Category information is therefore moved to a separate table. Figure 2-2 shows the tables in Third Normal Form.

Any time the data is in Third Normal Form, it is already automatically in Second and First Normal Form. The whole process can therefore actually be accomplished less tediously than by going from form to form. Simply arrange the data so that the columns in each table, other than the primary key, are dependent only on the *whole primary key*. Third Normal Form is sometimes described as "the key, the whole key, and nothing but the key."

Navigating Through the Data

The bookshelf database is now in Third Normal Form. Figure 2-3 shows a sample of what these tables might contain. It's easy to see how these tables are related. You navigate from one to the other to pull out information on a particular author, based on the keys to each table. The primary key in each table is able to uniquely identify a single row. Choose Stephen Jay Gould, for instance, and you can readily discover his record in the AUTHOR table, because AuthorName is the primary key.

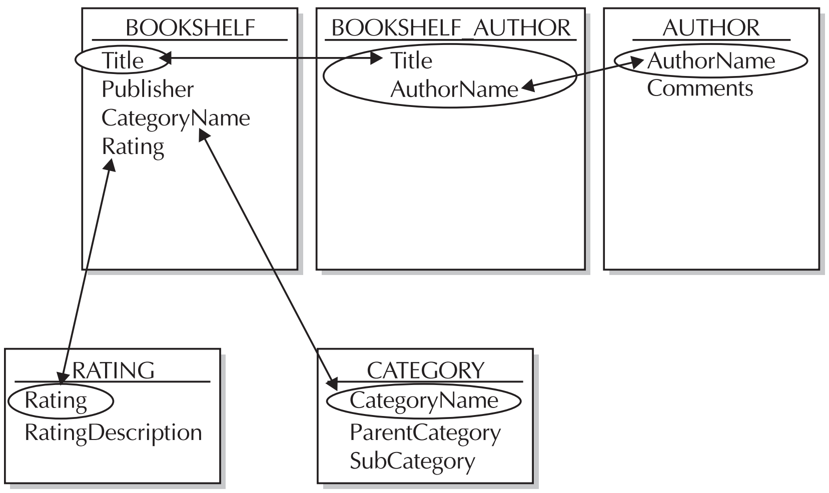

FIGURE 2-2. *BOOKSHELF and related tables*

```
AUTHOR
AuthorName            Comments
--------------------- -------------------------------------------
DIETRICH BONHOEFFER   GERMAN THEOLOGIAN, KILLED IN A WAR CAMP
ROBERT BRETALL        KIERKEGAARD ANTHOLOGIST
ALEXANDRA DAY         AUTHOR OF PICTURE BOOKS FOR CHILDREN
STEPHEN JAY GOULD     SCIENCE COLUMNIST, HARVARD PROFESSOR
SOREN KIERKEGAARD     DANISH PHILOSOPHER AND THEOLOGIAN
HARPER LEE            AMERICAN NOVELIST, PUBLISHED ONLY ONE NOVEL
LUCY MAUD MONTGOMERY  CANADIAN NOVELIST
JOHN ALLEN PAULOS     MATHEMATICS PROFESSOR
J. RODALE             ORGANIC GARDENING EXPERT

RATING
Rating     RatingDescription
---------- -----------------
1          ENTERTAINMENT
2          BACKGROUND INFORMATION
3          RECOMMENDED
4          STRONGLY RECOMMENDED
5          REQUIRED READING

CATEGORY
CategoryName   ParentCategory  SubCategory
-------------- --------------- ------------
ADULTREF       ADULT           REFERENCE
ADULTFIC       ADULT           FICTION
ADULTNF        ADULT           NONFICTION
CHILDRENPIC    CHILDREN        PICTURE BOOK
CHILDRENFIC    CHILDREN        FICTION
CHILDRENNF     CHILDREN        NONFICTION

BOOKSHELF_AUTHOR
Title                          AuthorName
------------------------------ ---------------------
TO KILL A MOCKINGBIRD          HARPER LEE
WONDERFUL LIFE                 STEPHEN JAY GOULD
INNUMERACY                     JOHN ALLEN PAULOS
KIERKEGAARD ANTHOLOGY          ROBERT BRETALL
KIERKEGAARD ANTHOLOGY          SOREN KIERKEGAARD
ANNE OF GREEN GABLES           LUCY MAUD MONTGOMERY
GOOD DOG, CARL                 ALEXANDRA DAY
LETTERS AND PAPERS FROM PRISON DIETRICH BONHOEFFER
```

FIGURE 2-3. *Sample data from the BOOKSHELF tables*

```
BOOKSHELF
Title                          Publisher            CategoryName Rating
------------------------------ -------------------- ------------ ------
TO KILL A MOCKINGBIRD          HARPERCOLLINS        ADULTFIC     5
WONDERFUL LIFE                 W.W.NORTON & CO.     ADULTNF      5
INNUMERACY                     VINTAGE BOOKS        ADULTNF      4
KIERKEGAARD ANTHOLOGY          PRINCETON UNIV PR    ADULTREF     3
ANNE OF GREEN GABLES           GRAMMERCY            CHILDRENFIC  3
GOOD DOG, CARL                 LITTLE SIMON         CHILDRENPIC  1
LETTERS AND PAPERS FROM PRISON SCRIBNER             ADULTNF      4
```

FIGURE 2-3. *Sample data from the BOOKSHELF tables* (continued)

Look up Harper Lee in the AuthorName column of the BOOKSHELF_AUTHOR table and you'll see that she has published one novel, whose title is "To Kill A Mockingbird." You can then check the publisher, category, and rating for that book in the BOOKSHELF table. You can check the RATING table for a description of the rating.

When you looked up "To Kill A Mockingbird" in the BOOKSHELF table, you were searching by the primary key for the table. To find the author of that book, you could reverse your earlier search path, looking through BOOKSHELF_AUTHOR for the records that have that value in the Title column—the column 'Title' is a foreign key in the BOOKSHELF_AUTHOR table. When the primary key for BOOKSHELF appears in another table, as it does in the BOOKSHELF_AUTHOR table, it is called a *foreign key* to that table.

These tables also show real-world characteristics: There are ratings and categories that are not yet used by books on the bookshelf. Because the data is organized logically, you can keep a record of potential categories, ratings, and authors even if none of the current books use those values.

This is a sensible and logical way to organize information, even if the "tables" are written in a ledger book or on scraps of paper in cigar boxes. Of course, there is still some work to do to turn this into a real database. For instance, AuthorName probably ought to be broken into FirstName and LastName, and you might want to find a way to show which author is the primary author, or if one is an editor rather than an author.

This whole process is called normalization. It really isn't any trickier than this. There are some other issues involved in a good design, but the basics of analyzing the "normal" relationships among the various elements of data are just as simple and straightforward as they've just been explained. It makes sense regardless of whether or not a relational database or a computer is involved at all.

One caution needs to be raised, however. Normalization is a part of the process of analysis. It is not design. Design of a database application includes many other considerations, and it is a fundamental mistake to believe that the normalized tables of the logical model are the "design" for the actual database. This fundamental confusion of analysis and design contributes to the stories in the press about the failure of major relational applications. These issues are addressed for developers more fully later in this chapter.

English Names for Tables and Columns

Once the relationships between the various elements of the data in an application are understood and the data elements are segregated appropriately, considerable thought must be devoted to choosing names for the tables and columns into which the data will be placed. This is an area given too little attention, even by those who should know better. Table and column names are often developed without consulting end users and without rigorous review. Both of these failings have serious consequences when it comes to actually using an application.

For example, consider the tables shown in Figure 2-3. The table and column names are virtually all self-evident. An end user, even one new to relational ideas and SQL, would have little difficulty understanding or even replicating a query such as this:

```
select Title, Publisher
  from BOOKSHELF
 order by Publisher;
```

Users understand this because the words are all familiar. There are no obscure or ill-defined terms. When tables with many more columns in them must be defined, naming the columns can be more difficult, but a few consistently enforced rules will help immensely. Consider some of the difficulties commonly caused by lack of naming conventions. What if you had chosen these names instead?

```
BOOKSHELF      B_A       AUTHS         CATEGORIES
-------------  --------  ------------  --------------
title          title     anam          cat
pub            anam      comms         p_cat
cat                                    s_cat
rat
```

The naming techniques in this table, as bizarre as they look, are unfortunately very common. They represent tables and columns named by following the conventions (and lack of conventions) used by several well-known vendors and developers.

Here are a few of the more obvious difficulties in the list of names:

- *Abbreviations are used without good reason.* This makes remembering the “spelling” of a table or column name virtually impossible. The names may as well be codes, because the users will have to look them up.
- *Abbreviations are inconsistent.*
- *The purpose or meaning of a column or table is not apparent from the name.* In addition to abbreviations making the spelling of names difficult to remember, they obscure the nature of the data the column or table contains. What is P_cat? Comms?
- *Underlines are used inconsistently.* Sometimes they are used to separate words in a name, but other times they are not. How will anyone remember which name does or doesn’t have an underline?
- *Use of plurals is inconsistent.* Is it CATEGORY or CATEGORIES? Comm or Comms?

- *Rules apparently used have immediate limitations.* If the first letter of the table name is to be used for a name column, as in Anam for a table whose table name starts with 'A', what happens when a second table beginning with the letter 'A' becomes necessary? Does the name column in that table also get called ANam? If so, why isn't the column in both simply called Name?

These are only a few of the most obvious difficulties. Users subjected to poor naming of tables and columns will not be able to simply type English queries. The queries won't have the intuitive and familiar "feel" that the BOOKSHELF table query has, and this will harm the acceptance and usefulness of the application significantly.

Programmers used to be required to create names that were a maximum of six to eight characters in length. As a result, names unavoidably were confused mixes of letters, numbers, and cryptic abbreviations. Like so many other restrictions forced on users by older technology, this one is just no longer applicable. Oracle allows table and column names up to 30 characters long. This gives designers plenty of room to create full, unambiguous, and descriptive names.

The difficulties outlined here imply solutions, such as avoiding abbreviations and plurals, and either eliminating underlines or using them consistently. These quick rules of thumb will go a long way in solving the naming confusion so prevalent today. At the same time, naming conventions need to be simple, easily understood, and easily remembered. In a sense, what is called for is a normalization of names. In much the same way that data is analyzed logically, segregated by purpose, and thereby normalized, the same sort of logical attention needs to be given to naming standards. The job of building an application is improperly done without it.

English Words for the Data

Having raised the important issue of naming conventions for tables and columns, the next step is to look at the data itself. After all, when the data from the tables is printed on a report, how self-evident the data is will determine how understandable the report is. In the BOOKSHELF example, Rating is a code value, and Category is a concatenation of multiple values. Is this an improvement? If you asked another person about a book, would you want to hear that it was a rated a 4 in AdultNF? Why should a machine be permitted to be less clear?

Additionally, keeping the information in English makes writing and understanding queries much simpler. The query should be as English-like as possible:

```
select Title, AuthorName
  from BOOKSHELF_AUTHOR;

Title                          AuthorName
------------------------------ ---------------------
TO KILL A MOCKINGBIRD          HARPER LEE
WONDERFUL LIFE                 STEPHEN JAY GOULD
INNUMERACY                     JOHN ALLEN PAULOS
KIERKEGAARD ANTHOLOGY          ROBERT BRETALL
KIERKEGAARD ANTHOLOGY          SOREN KIERKEGAARD
ANNE OF GREEN GABLES           LUCY MAUD MONTGOMERY
GOOD DOG, CARL                 ALEXANDRA DAY
LETTERS AND PAPERS FROM PRISON DIETRICH BONHOEFFER
```

Capitalization in Names and Data

Oracle makes it slightly easier to remember table and column names by ignoring whether you type in capital letters, small letters, or a mixture of the two. It stores table and column names in its internal data dictionary in uppercase. When you type a query, it instantly converts the table and column names to uppercase, and then checks for them in the dictionary. Some other relational systems are case-sensitive. If users type a column name as "Ability," but the database thinks it is "ability" or "ABILITY" (depending on what it was told when the table was created), it will not understand the query.

NOTE
You can force Oracle to create tables and columns with mixed-case names, but doing so will make querying and working with the data difficult. Use the default uppercase behavior.

The ability to create case-sensitive table names is promoted as a benefit because it allows programmers to create many tables with, for instance, similar names. They can make a worker table, a Worker table, a wORker table, and so on. These will all be separate tables. How is anyone, including the programmer, supposed to remember the differences? This is a drawback, not a benefit, and Oracle was wise not to fall into this trap.

A similar case can be made for data stored in a database. There are ways to find information from the database regardless of whether the data is in uppercase or lowercase, but these methods impose an unnecessary burden. With few exceptions, such as legal text or form-letter paragraphs, it is much easier to store data in the database in uppercase. It makes queries easier and provides a more consistent appearance on reports. When and if some of this data needs to be put into lowercase, or mixed uppercase and lowercase (such as the name and address on a letter), then the Oracle functions that perform the conversion can be invoked. It will be less trouble overall, and less confusing, to store and report data in uppercase.

Looking back over this chapter, you'll see that this practice was not followed. Rather, it was delayed until the subject could be introduced and put in its proper context. From here on, with the exception of one or two tables and a few isolated instances, data in the database will be in uppercase.

Normalizing Names

Several query tools have come on the market whose purpose is to let you make queries using common English words instead of odd conglomerations. These products work by building a logical map between the common English words and the hard-to-remember, non-English column names, table names, and codes. The mapping takes careful thought, but once completed, it makes the user's interaction with the application easy. Why not put the care in at the beginning? Why create a need for yet another layer, another product, and more work, when much of the confusion can be avoided simply by naming things better the first time around?

For performance reasons, it may be that some of an application's data must still be stored in a coded fashion within the computer's database. These codes should *not* be exposed to users, either during data entry or retrieval, and Oracle allows them to be easily hidden.

The instant that data entry requires codes, key-entry errors increase. When reports contain codes instead of English, errors of interpretation begin. And when users need to create new or ad hoc reports, their ability to do so quickly and accurately is severely impaired both by codes and by not being able to remember strange column and table names.

Oracle gives users the power to see and work with English throughout the entire application. It is a waste of Oracle's power to ignore this opportunity, and it will without question produce a less understandable and less productive application. Developers should seize the opportunity. Users should demand it. Both will benefit immeasurably.

Good Design Has a Human Touch

At this point, if you are new to Oracle you may want to go right into working with Oracle and the SQL language. That is covered in the next chapter; the remainder of this chapter focuses on performance, naming, and design considerations. You can refer back to this section later when you are ready to design and implement an application.

This section of this chapter looks at a method of approaching a development project that takes into account the real business tasks your end users have to accomplish. This distinguishes it from the more common data orientation of many developers and development methodologies. Data normalization and CASE (Computer Aided Software Engineering) technologies have become so much the center of attention with relational application development that a focus on the data and the issues of referential integrity, keys, normalization, and table diagrams has become almost an obsession. They are so often confused with design—and even believed to *be* design—that the reminder that they are analysis is often met with surprise.

Normalization is analysis, not design. And it is only a part of the analysis necessary to understand a business and build a useful application. The goal of application development, after all, is to help the business run more successfully by improving the speed and efficiency with which business tasks are done and by making the environment in which people work as meaningful and supportive as possible. Give people control over their information, and intuitive, straightforward access to it, and they will respond gratefully and productively. Assign the control to a remote group, cloud the information in codes and user-hostile interfaces, and they will be unhappy and unproductive.

The methods outlined in this section are not intended to be a rigorous elucidation of the process, and the tools you use and are familiar with for data structures or flows are probably sufficient for the task. The purpose here is to disclose an approach that is effective in creating responsive, appropriate, and accommodating applications.

Understanding the Application Tasks

One of the often-neglected steps in building software is understanding the end user's job—the tasks that computer automation is intended to support. Occasionally, this is because the application itself is quite specialized; more often, it is because the approach to design tends to be data-oriented. Frequently, these are the major questions asked in the analysis:

- What data should be captured?
- How should the data be processed?
- How should the data be reported?

These questions expand into a series of subquestions, and include issues such as input forms, codes, screen layouts, computations, postings, corrections, audit trails, retention, storage volumes, processing cycles, report formatting, distribution, and maintenance. These are all vitally important areas. One difficulty, however, is that they all focus solely on data.

People *use* data, but they *do* tasks. One might argue that while this may be true of professional workers, key-entry clerks really only transfer data from an input form to a keyboard; their tasks are very data-oriented. This is a fair portrayal of these jobs today. But is this a consequence of the real job that needs to get done, or is it a symptom of the design of the computer application? Using humans as input devices, particularly for data that is voluminous, consistent in format (as on forms), and in a limited range of variability, is an expensive and antiquated, not to mention dehumanizing, method of capturing data. Like the use of codes to accommodate machine limitations, it's an idea whose time has passed.

This may sound like so much philosophy, but it has practical import in the way application design is done. People use data, but they do tasks. And they don't do tasks through to completion one at a time. They do several tasks that are subsets of or in intersection with each other, and they do them all at once, in parallel.

When designers allow this idea to direct the analysis and creation of an application, rather than focusing on the data orientation that has been historically dominant, the very nature of the effort changes significantly. Why have windowing environments been so successful? Because they allow a user to jump quickly between small tasks, keeping them all active without having to shut down and exit one in order to begin another. The windowing environment comes closer to mapping the way people really think and work than the old "one thing at a time" approach ever did. This lesson should not be lost. It should be built upon.

Understanding the application tasks means going far beyond identifying the data elements, normalizing them, and creating screens, processing programs, and reports. It means really understanding what the users do and what their tasks are, and designing the application to be responsive to those tasks, not just to capture the data associated with them. In fact, when the orientation is toward the data, the resulting design will inevitably *distort* the users' tasks rather than support them.

How do you design an application that is responsive to tasks rather than data? The biggest hurdle is simply understanding that focusing on tasks is necessary. This allows you to approach the analysis of the business from a fresh perspective.

The first step in the analysis process is to understand the tasks. For which tasks do the members of this group really need to use computers? What is the real service or product produced? This seems like a fundamental and even simplistic first question, but you'll find that a surprising number of businesspeople are quite unclear about the answer. An amazing number of businesses, from healthcare to banking, from shipping to manufacturing, used to think they were in the data processing business. After all, they input data, process it, and report it, don't they? This delusion is yet another symptom of the data orientation our systems designs have had that has led dozens of companies to attempt to market their imagined "real" product, data processing, with disastrous consequences for most of them.

Hence the importance of learning about a business application: You have to keep an open mind, and may often have to challenge pet notions about what the business is in order to learn what it really is. This is a healthy, if sometimes difficult, process.

And, just as it is essential that businesspeople become literate users of SQL and understand the basics of the relational model, so it is important that application designers really understand the service or product being delivered, and the tasks *necessary* to make that happen. A project team that includes end users who have been introduced to the essentials of SQL and the relational approach, such as by reading this book, and designers who are sensitive to end users' needs and understand the value of a task-oriented, readable application environment, will turn out extraordinarily good systems. The members of such a project team check, support, and enhance each other's efforts.

One approach to this process is to develop two converging documents: a task document and a data document. It is in the process of preparing the task document that the deep understanding of the application comes about. The data document will help implement the vision and ensure that the details and rules are all accounted for, but the task document defines the vision of what the business is.

Outline of Tasks

The task document is a joint effort of the business users and the application designers. It lists the tasks associated with the business from the top down. It begins with a basic description of the business. This should be a simple declarative sentence of three to ten words, in the active voice, without commas and with a minimum of adjectives:

> We sell insurance.

It should not be:

> Amalgamated Diversified is a leading international supplier of financial resources, training, information processing, transaction capture and distribution, communications, customer support, and industry direction in the field of shared risk funding for health care maintenance, property preservation, and automobile liability.

There is a tremendous temptation to cram every little detail about a business and its dreams about itself into this first sentence. Don't do it. The effort of trimming the descriptive excesses down to a simple sentence focuses the mind wonderfully. If you can't get the business down to ten words, you haven't understood it yet.

But, as an application designer, creating this sentence isn't your task alone; it is a joint effort with the business user, and it initiates the task documentation process. It provides you with the opportunity to begin serious questioning about what the business does and how it does it. This is a valuable process for the business itself, quite independent of the fact that an application is being built. You will encounter numerous tasks and subtasks, procedures, and rules that will prove to be meaningless or of marginal use. Typically, these are artifacts of either a previous problem, long since solved, or of information or reporting requests from managers long since departed.

Some wags have suggested that the way to deal with too many reports being created, whether manually or by computer, is to simply stop producing them and see if anyone notices. This is a humorous notion, but the seed of truth it contains needs to be a part of the task documentation process. In fact, it proved quite useful in Y2K remediation efforts—many programs and reports didn't have to be fixed, simply because they were no longer used!

Your approach to the joint effort of documenting tasks allows you to ask skeptical questions and look at (and reevaluate the usefulness of) what may be mere artifacts. Be aware, however, that you need to proceed with the frank acknowledgment that you, as a designer, cannot understand the business as thoroughly as the user does. There is an important line between seizing the opportunity of an application development to rationalize what tasks are done and why, and possibly offending the users by presuming to understand the "real" business better than they do.

Ask the user to describe a task in detail and explain to you the reason for each step. If the reason is a weak one, such as "We've always done it this way," or "I think they use this upstairs for something," red flags should go up. Say that you don't understand, and ask again for an explanation. If the response is still unsatisfactory, put the task and your question on a separate list for resolution. Some of these will be answered simply by someone who knows the subject better, others will require talking to senior management, and many tasks will end up eliminated because they are no longer needed. One of the evidences of a good analysis process is the improvement of existing procedures, independent of, and generally long before, the implementation of a new computer application.

General Format of the Task Document

This is the general format for the task document:

- Summary sentence describing the business (three to ten words)
- Summary sentences describing and numbering the major tasks of the business (short sentences, short words)
- Additional levels of task detail, as needed, within each of the major tasks

By all means, follow the summary sentence for every level with a short, descriptive paragraph, if you want, but don't use this as an excuse to avoid the effort of making the summary sentence clear and crisp. Major tasks are typically numbered 1.0, 2.0, 3.0, and so on, and are sometimes referred to as zero-level tasks. The levels below each of these are numbered using additional dots, as in 3.1 and 3.1.14. Each major task is taken down to the level where it is a collection of *atomic tasks*—tasks for which no subtask is meaningful in itself and that, once started, is either taken to completion or dropped entirely. Atomic tasks are never left half-finished.

Writing a check is an atomic task; filling in the dollar amount is not. Answering the telephone as a customer service representative is not an atomic task; answering the phone and fulfilling the customer's request is atomic. Atomic tasks must be meaningful and must complete an action.

The level at which a task is atomic will vary by task. The task represented by 3.1.14 may be atomic yet still have several additional sublevels. The task 3.2 may be atomic, or 3.1.16.4 may be. What is important is not the numbering scheme (which is nothing more than a method for outlining a hierarchy of tasks) but the decomposition to the atomic level. The atomic tasks are the fundamental building blocks of the business. Two tasks can still be atomic if one occasionally depends upon the other, but only if each can and does get completed independently. If two tasks always depend upon each other, they are not atomic. The real atomic task includes them both.

In most businesses, you will quickly discover that many tasks do not fit neatly into just one of the major (zero-level) tasks, but seem to span two or more, and work in a network or "dotted line" fashion. This is nearly always evidence of improper definition of the major tasks or incomplete atomization of the lower tasks. The goal is to turn each task into a conceptual "object," with a well-defined idea of what it does (its goal in life) and what resources (data, computation, thinking, paper, pencil, and so on) it uses to accomplish its goal.

Insights Resulting from the Task Document

Several insights come out of the task document. First, because the task document is task-oriented rather than data-oriented, it is likely to substantially change the way user screens are designed. It will affect what data is captured, how it is presented, how help is implemented, and how users switch from one task to another. The task orientation will help ensure that the most common kinds of jumping between tasks will not require inordinate effort from the user.

Second, the categorization of major tasks will change as conflicts are discovered; this will affect how both the designers and the business users understand the business.

Third, even the summary sentence itself will probably change. Rationalizing a business into atomic task "objects" forces a clearing out of artifacts, misconceptions, and unneeded dependencies that have long weighed down the business unnecessarily.

This is not a painless process, but the benefits in terms of the business's self-understanding, the cleanup of procedures, and the automation of the tasks will usually far exceed the emotional costs and time spent. It helps immensely if there is general understanding going into the project that uncomfortable questions will be asked, incorrect assumptions corrected, and step-by-step adjustments made to the task document until it is completed.

Understanding the Data

In conjunction with the decomposition and description of the tasks, the resources required at each step are described in the task document, especially in terms of the data required. This is done on a task-by-task basis, and the data requirements are then included in the data document. This is a conceptually different approach from the classical view of the data. You will not simply take the forms and screens currently used by each task and record the elements they contain. The flaw in this "piece of paper in a cigar box" approach lies in our tendency (even though we don't like to admit it) to accept anything printed on paper as necessary or true.

In looking at each task, you should determine what data is necessary to do the task, rather than what data elements are on the form you use to do the task. By requiring that the definition of the data needed come from the task rather than from any existing forms or screens, you force an examination of the true purpose of the task and the real data requirements. If the person doing the task doesn't know the use to which data is put, the element goes on the list for resolution. An amazing amount of garbage is eliminated by this process.

Once the current data elements have been identified, they must be carefully scrutinized. Numeric and letter codes are always suspect. They disguise real information behind counterintuitive, meaningless symbols. There are times and tasks for which codes are handy, easily remembered, or made necessary by sheer volume. But, in your final design, these cases should be rare and obvious. If they are not, you've lost your way.

In the scrutiny of existing data elements, codes should be set aside for special attention. In each case, ask yourself whether the element should be a code. Its continued use as a code should be viewed suspiciously. There must be good arguments and compelling reasons for perpetuating the disguise. The process for converting codes back into English is fairly simple, but is a joint effort. The codes are first listed, by data element, along with their meanings. These are then examined by users and designers, and short English versions of the meanings are proposed, discussed, and tentatively approved.

In this same discussion, designers and end users should decide on names for the data elements. These will become column names in the database, and will be regularly used in English queries, so the names should be descriptive (avoiding abbreviations, other than those common to the business) and singular. Because of the intimate relationship between the column name and the data it contains, the two should be specified simultaneously. A thoughtful choice of a column name will vastly simplify determining its new English contents.

Data elements that are not codes also must be rigorously examined. By this point, you have good reason to believe that all of the data elements you've identified are necessary to the business tasks, but they are not necessarily well-organized. What appears to be one data element in the existing task may in fact be several elements mixed together that require separation. Names, addresses, and phone numbers are very common examples of this, but every application has a wealth of others.

First and last names were mixed together, for example, in the AUTHOR table. The AuthorName column held both first and last names, even though the tables were in Third Normal Form. This would be an extremely burdensome way to actually implement an application, in spite of the fact that the normalization rules were technically met. To make the application practical and prepare it for English queries, the AuthorName column needs to be decomposed into at least two new columns, LastName and FirstName. This same categorization process is regularly needed in rationalizing other data elements, and is often quite independent of normalization.

The degree of decomposition depends on how the particular data elements are likely to be used. It is possible to go much too far and decompose categories that, though made up of separable pieces, provide no additional value in their new state. Decomposition is application-dependent on an element-by-element basis. Once decomposition has been done, these new elements, which will become columns, need to be thoughtfully named, and the data they will contain needs to be scrutinized. Text data that will fall into a definable number of values should be reviewed for naming. These column names and values, like those of the codes, are tentative.

The Atomic Data Models

Now the process of normalization begins, and with it the drawing of the atomic data models. There are many good texts on the subject and a wide variety of analysis and design tools that can speed the process, so this book doesn't suggest any particular method, since recommending one method may hinder rather than help.

Each atomic transaction should be modeled, and should be labeled with the task number to which it applies. Included in the model are table names, primary and foreign keys, and major columns. Each normalized relationship should have a descriptive name, and estimated row counts and transaction rates should appear with each table. Accompanying each model is an additional sheet with all of the columns and datatypes, their ranges of value, and the tentative names for the tables, columns, and named values in the columns.

The Atomic Business Model

This data document is now combined with the task document. The combined document is a business model. It's reviewed jointly by the application designers and end users for accuracy and completeness.

The Business Model

At this point, both the application designers and the end users should possess a clear vision of the business, its tasks, and its data. Once the business model is corrected and approved, the process of synthesizing the tasks and data models into an overall business model begins. This part of the process sorts common data elements between tasks, completes final, large-scale normalization, and resolves consistent, definitive names for all of the parts.

This can be quite a large drawing for major applications, with supporting documentation that includes the tasks, the data models (with corrected element names, based on the full model), and a list of each of the full-scale tables and their column names, datatypes, and contents. A final check of the effort is made by tracing the data access paths of each transaction in the full business model to determine that all the data the transaction requires is available for selection or insertion, and that no tasks insert data with elements missing that are essential to the model's referential integrity.

With the exception of the effort spent to properly name the various tables, columns, and common values, virtually everything to this point has been analysis, not design. The aim has been to promote understanding of the business and its components.

Data Entry

Screen design does not proceed from the business model. It is not focused on tables, but rather on tasks, so screens are created that support the task orientation and the need to jump between subtasks when necessary. In practical terms, this will often map readily to a primary table used by the task, and to other tables that can be queried for values or updated as the primary table is accessed.

But there will also be occasions where there simply is no main table, but instead a variety of related tables, all of which will supply or receive data to support the task. These screens will look and act quite differently from the typical table-oriented screens developed in many applications, but they will significantly amplify the effectiveness of their users and their contribution to the business. And that's the whole purpose of this approach.

The interaction between the user and the machine is critical; the input and query screens should consistently be task-oriented and descriptive, in English. The use of icons and graphical interfaces plays an important role as well. Screens must reflect the way work is actually done, and be built to respond in the language in which business is conducted.

Query and Reporting

If anything sets apart the relational approach, and SQL, from more traditional application environments, it is the ability for end users to easily learn and execute ad hoc queries. These are those reports and one-time queries that are outside of the basic set usually developed and delivered along with the application code.

With SQLPLUS (and other reporting tools), end users are given unprecedented control over their own data. Both the users and developers benefit from this ability: the users, because they can build reports, analyze information, modify their queries, and reexecute them all in a matter of minutes, and the developers, because they are relieved of the undesirable requirement of creating new reports.

Users are granted the power to look into their data, analyze it, and respond with a speed and thoroughness unimaginable just a few years ago. This leap in productivity is greatly extended if the tables, columns, and data values are carefully crafted in English; it is greatly foreshortened if bad naming conventions and meaningless codes and abbreviations are permitted to infect the design. The time spent in the design process to name the objects consistently and descriptively will pay off quickly for the users, and therefore for the business.

Some people, typically those who have not built major relational applications, fear that turning query facilities over to end users will cripple the machine on which the facilities are used. The fear is that users will write inefficient queries that will consume overwhelming numbers of CPU cycles, slowing the machine and every other user. Experience shows that this generally is not true. Users quickly learn which kinds of queries run fast, and which do not. Further, most business intelligence and reporting tools available today can estimate the amount of time a query will take, and restrict access—by user, time of day, or both—to queries that would consume a disproportionate amount of resources. In practice, the demands users make on a machine only occasionally get out of hand, but the benefits they derive far exceed the cost of the processing. Virtually any time you can move effort from a person to a machine, you save money.

The real goal of design is to clarify and satisfy the needs of the business and business users. If there is a bias, it must always be toward making the application easier to understand and use, particularly at the expense of CPU or disk, but less so if the cost is an internal complexity so great that maintenance and change become difficult and slow.

Toward Object Name Normalization

The basic approach to naming is to choose meaningful, memorable, and descriptive readable names, avoiding abbreviations and codes, and using underlines either consistently or not at all. In a large application, table, column, and data names will often be multiword, as in the case of ReversedSuspenseAccount or Last_GL_Close_Date. The goal of thoughtful naming methods is ease of use: The names must be easily remembered and must follow rules that are easily explained and applied. In the pages ahead, a somewhat more rigorous approach to naming is presented, with the ultimate goal of developing a formal process of object name normalization.

Level-Name Integrity

In a relational database system, the hierarchy of objects ranges from the database, to the table owners, to the tables, to the columns, to the data values. In very large systems, there may even be multiple databases, and these may be distributed within locations. For the sake of brevity, the higher levels will be ignored for now, but what is said will apply to them as well.

Each level in this hierarchy is defined within the level above it, and each level should be given names appropriate to its own level and should not incorporate names from outside its own level. For example, a table cannot have two columns called Name, and the account named George cannot own two tables named AUTHOR.

There is no requirement that each of George's tables have a name that is unique throughout the entire database. Other owners may have AUTHOR tables as well. Even if George is granted access to them, there is no confusion, because he can identify each table uniquely by prefixing its owner's name to the table name, as in Dietrich.AUTHOR. It would not be logically consistent to incorporate George's owner name into the name of each of his tables, as in GEOAUTHOR, GEOBOOKSHELF, and so on. This confuses and complicates the table name by placing part of its parent's name in its own, in effect a violation of *level-name integrity.*

Brevity should never be favored over clarity. Including pieces of table names in column names is a bad technique, because it violates the logical idea of levels, and the level-name integrity that this requires. It is also confusing, requiring users to look up column names virtually every time they want to write a query. Object names must be unique within their parent, but no incorporation of names from outside an object's own level should be permitted.

The support for abstract datatypes in Oracle strengthens your ability to create consistent names for attributes. If you create a datatype called ADDRESS_TY, it will have the same attributes each time it is used. Each of the attributes will have a consistent name, datatype, and length, making their implementation more consistent across the enterprise. However, using abstract datatypes in this manner requires that you do both of the following:

- Properly define the datatypes at the start so that you can avoid the need to modify the datatype later
- Support the syntax requirements of abstract datatypes (see Chapter 4 for details)

Foreign Keys

The one area of difficulty with using brief column names is the occasional appearance of a foreign key in a table in which another column has the same name that the foreign key column has in its home table. One possible long-term solution is to allow the use of the full foreign key name, including the table name of its home table, as a column name in the local table (such as BOOKSHELF.Title as a column name).

The practical need to solve the same-name column problem requires one of the following actions:

- Invent a name that incorporates the source table of the foreign key in its name without using the dot (using an underline, for instance).
- Invent a name that incorporates an abbreviation of the source table of the foreign key in its name.
- Invent a name different from its name in its source table.
- Change the name of the conflicting column.

None of these is particularly attractive, but if you come across the same-name dilemma, you'll need to take one of these actions.

Singular Names

One area of great inconsistency and confusion is the question of whether objects should have singular or plural names. Should it be the AUTHOR table or the AUTHORS table? Should it be the Name column or the Names column?

There are two helpful ways to think about this issue. First, consider some columns common to nearly every database: Name, Address, City, State, and Zip. Other than the first column, does it ever occur to anyone to make these names plural? It is nearly self-evident when considering these names that they describe the contents of a single row, a record. Even though relational databases are "set-oriented," clearly the fundamental unit of a set is a row, and it is the content of that row that is well-described by singular column names. When designing a data entry screen to capture a person's name and address, should it look like this?

```
Names: _______________________________________________________

Addresses: ___________________________________________________

Cities: ____________________________ States __ Zips _____-____
```

Or will you make these column names singular on the screen, since you're capturing *one* name and address at a time, but tell the users that when they write queries they must all be converted to plural? It is simply more intuitive and straightforward to restrict column names to singular.

If all objects are named consistently, neither you nor a user has to try to remember the rules for what is plural and what isn't. The benefit of this should be obvious. Suppose we decide that all objects will henceforth be plural. We now have an "s" or an "es" on the end of virtually every object, perhaps even on the end of each word in a long multiword object name. Of what possible benefit is it to key all of these extra letters all the time? Is it easier to use? Is it easier to understand? Is it easier to remember? Obviously, it is none of these.

Therefore, the best solution is this: All object names are always singular. The sole exception to this rule is any widely accepted term already commonly used in the business, such as "sales."

Brevity

As mentioned earlier, clarity should never be sacrificed for brevity, but given two equally meaningful, memorable, and descriptive names, always choose the shorter. During application development, propose alternative column and table names such as these to a group of users and developers and get their input on choosing the clearest name. How do you build lists of alternatives? Use a thesaurus and a dictionary. On a project team dedicated to developing superior, productive applications, every team member should be given a thesaurus and a dictionary as basic equipment, and then should be reminded over and over again of the importance of careful object naming.

Object Name Thesaurus

Ultimately, relational databases should include an object name thesaurus, just as they include a data dictionary. This thesaurus should enforce the company's naming standards and ensure consistency of name choice and abbreviation (where used).

Such standards may require the use of underlines in object naming to make the parsing of the name into component parts a straightforward task. This also helps enforce the consistent use of underlines, rather than the scattered, inconsistent usage within an application that underlines frequently receive now.

If you work directly with a government agency or large firm, that organization may already have object-naming standards. The object-naming standards of large organizations have over the years radiated into the rest of the commercial marketplace, and may form the basis for the naming standards used at your company. For example, those standards may provide the direction to choose between "Corporation" and "Firm." If they do not, you should develop your naming standards to be consistent both with those base standards and with the guidelines put forth in this chapter.

Intelligent Keys and Column Values

Intelligent keys are so named because they contain nontrivial combinations of information. The term is misleading in the extreme because it implies something positive or worthwhile. A more meaningful term might be "overloaded" keys. General ledger and product codes often fall into this category and contain all the difficulties associated with other codes, and more. Further, the difficulties found in overloaded keys also apply to non-key columns that are packed with more than one piece of meaningful data.

Typical of an overloaded key or column value is this description: "The first character is the region code. The next four characters are the catalog number. The final digit is the cost center code, unless this is an imported part, in which case an *I* is tagged onto the end of the number, or unless it is a high-volume item, such as screws, in which case only three digits are used for catalog number, and the region code is HD."

Eliminating overloaded key and column values is essential in good relational design. The dependencies built on pieces of these keys (usually foreign keys into other tables) are all at risk if the structure is maintained. Unfortunately, many application areas have overloaded keys that have been used for years and are deeply embedded in the company's tasks. Some of them were created during earlier efforts at automation, using databases that could not support multiple key columns for composite keys. Others came about through historical accretion, by forcing a short code, usually numeric, to mean more and to cover more cases than it was ever intended to at the beginning. Eliminating the existing overloaded keys may have practical ramifications that make it impossible to do immediately. This makes building a new, relational application more difficult.

The solution to this problem is to create a new set of keys, both primary and foreign, that properly normalizes the data; then, make sure that people can access tables only through these new keys. The overloaded key is then kept as an additional, and unique, table column. Access to it is still possible using historical methods (matching the overloaded key in a query, for instance), but the newly structured keys are promoted as the preferred method of access. Over time, with proper training, users will gravitate to the new keys. Eventually, the overloaded keys (and other overloaded column values) can simply be **NULL**ed out or dropped from the table.

Failing to eliminate overloaded keys and values makes extracting information from the database, validating the values, assuring data integrity, and modifying the structure all extremely difficult and costly.

The Commandments

All of the major issues in designing for productivity have now been discussed. It probably is worthwhile to sum these up in a single place—thus "The Commandments" (or perhaps "The Suggestions"). Their presentation does not assume that you need to be told what to do, but rather that you are capable of making rational judgments and can benefit from the experience of others facing the same challenges. The purpose here is not to describe the development cycle, which you probably understand better than you want to, but rather to bias that development with an orientation that will radically change how the application will look, feel, and be used. Careful attention to these ideas can dramatically improve the productivity and happiness of an application's users.

The Ten Commandments of Humane Design

1. Include users. Put them on the project team and teach them the relational model and SQL.
2. Name tables, columns, keys, and data jointly with the users. Develop an application thesaurus to ensure name consistency.
3. Use English words that are meaningful, memorable, descriptive, short, and singular. Use underlines consistently or not at all.
4. Don't mix levels in naming.

5. Avoid codes and abbreviations.
6. Use meaningful keys where possible.
7. Decompose overloaded keys.
8. Analyze and design from the tasks, not just the data. Remember that normalization is not design.
9. Move tasks from users to the machine. It is profitable to spend cycles and storage to gain ease of use.
10. Don't be seduced by development speed. Take time and care in analyses, design, testing, and tuning.