ROC276

Introduction to Database Design

Ted Roche
Ted Roche & Associates, LLC
http://www.tedroche.com

If you're only ever going to have two salesman, why create a separate table for them? And if it's always going to be Bill and Bob, why not just include their names in the sales order? If the road to Hades is paved with good intentions, intentionally and unnecessarily violating the basic principles of normalization is the express lane. This session reviews the principles of data design analysis, normalization and naming conventions.

Introduction

A fellow developer once asked me “Is it possible to develop a good application from a bad data design?” What a good question! I pondered for a few moments before replying that yes, a good application could be created from a bad data design, but it was a lot more work to create and a lot more work to maintain than one created from a good data design.

So, here we are, lazy programmers all. Lazy is one of the ultimate compliments for a programmer: it means that they will find a way to do things efficiently once, so that they don't need to go back and fix things and they don't get stuck doing boring things over and over again. Good data design is the lazy programmer's way to ensure that the design is right up front, that it is flexible enough to support change, and that it is rugged enough to support the needs of the application.

Terminology

First, a quick terminology review for those new to database design.

A **field** is a single piece of information, such a person’s first name or an item’s price. Divide your information so that fields can stand on their own, and don’t need to be sub-divided when you are processing. For example, if your part number is composed of a part type, a region code, a rating and a sub-part number, such as AN-33-X4-1234, it is often be better to break that information into separate fields and combine it when needed, rather than to constantly split the field when looking for all parts from one region.

Each field has a single **datatype**. Datatypes specify that a field holds a particular kind of information; has upper and lower limits on their capacity; and are restricted on what kind of information they can hold. Fields may be character, integer, date, datetime (a combination of date and time), numeric, double, float, currency logical (true or false), memo (very long free-form text or binary data). Specialized datatypes exist to hold OLE information (general fields), Macintosh binary (picture fields), but are rarely used.

A collection of fields that holds a single piece of information forms a **record**. For example, you might record a check received from a customer as:

Field name	Type	Data
Customer Number	Integer	4321
Check Number	Integer	5678
Amount	Numeric (9,2)	$910.11
Date Received	Date	22 July 2001

A collection of these records would form a **table**. The table of data can be viewed in many ways, but the standard form used by the Fox BROWSE command looks like this:

Customer	Check	Amount	Date
1243	3121	232	01/01/1901
3232	43243	3343	02/02/2002
23232	42	43.34	03/03/1903

Figure 1: Why do you think they call them tables, rows and columns? Data is often viewed rectilinearly.

This geometry leads to other names for the items, records are often called **rows** and fields **columns**. A row or record is often more formally called a **tuple**.

A **key** is an expression composed of one or more fields that is used to link records from different tables together, to speed access to a record, or to temporarily sort records into a particular order. Keys are stored in **indexes.** Different physical implementations can be created in different database engines. Visual FoxPro supports stand-alone indexes (IDX extensions), compound indexes (CDX) and structural compound indexes (CDX).

A **primary key** uniquely identifies each record in a table. A **candidate key** is another key that could also uniquely identify a record, but it was not selected as the primary key. A **foreign key** is a primary key column duplicated in another table, where it links the table to its originating table.

A **relation** links two tables, primary key to foreign key. **Cardinality** describes the potential number of records in the relation. Since a primary key must be unique, one side of the relation is always one or zero. The other side can be zero, one or many. The "one" side is often called the **parent** and the other side the **child**. While there are different notations, most are pretty readable: in text, "one-to-many" is 1:M, "one-to-zero-or-many" is 1:0,M and "one-to-zero-or-one" is 1:0,1. See **Figure 2** for typical graphical representations of the different relations.

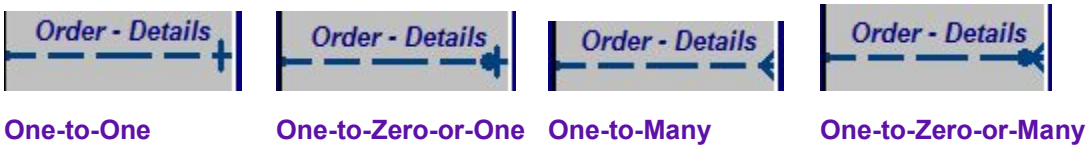

Figure 2: Graphical representation of the cardinality of relations.

It's a good practice to start out with a **logical database design**: a design that expresses the entities and relationships without regard for the practical limitations of the underlying database engine. This is following the principle of "First, get the design right, then make it practical." In reality, disk space and data engine performance issues make the need for a second **physical database design** often unnecessary.

Normalization

Premature optimization, whether the denormalization of data or the improper combining of entities, is a major source of database anguish.

The fundamentals of normalization are pretty easy, despite the sometimes imposing terminology. The purposes of normalization are to eliminate repetition in the data that takes up unnecessary space, forces us to do extra work, and can lead to misrepresented information. Chris Date cleverly summed it up years ago as "the Key, the whole Key, and nothing but the Key." Failing to properly normalize a data design results in repetitive code, difficultly in keeping the data model consistent, and incorrect query results. Intentional denormalization is often justified by claiming the performance would be unacceptable

(without testing the hypothesis) or that the coding would be too difficult. This well-intentioned *premature optimization*, whether the denormalization of data or the improper combining of entities, is a major source of database anguish. This section covers the textbook definitions of normalization with the usual simple examples, and then digs into some much tougher real-world examples.

First normal form: no repeating groups

Example: order table with Item1, Price1, Quantity1, Item2, Price2, Quantity2, etc.
Solution: move the individual lines of order detail to their own separate table.

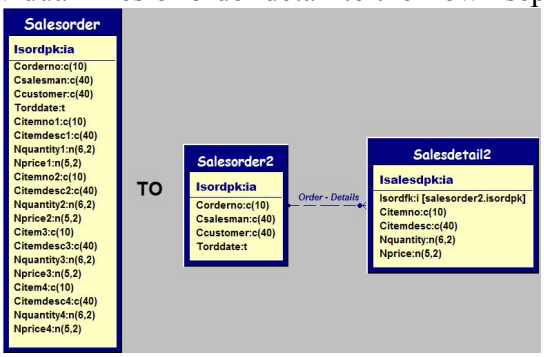

Figure 3: Reduce long, tall tables to one-to-many relationships by removing repeating groups to achieve first normal form.

Second normal form: full primary key required for each column

Note: I am an advocate (some would rightly say fanatic) for single-field, integer, non-data-bearing primary keys. There are a bunch of good reasons for this, covered later. However, for the discussion of the 2nd normal form, consider the "natural" primary key as those fields that make up unique identification of one record.

Example: An order detail record has fields of customer number, part number, part description, quantity and price – for this system, let's suppose the price can be changed either in the order detail or in the inventory table so that price is appropriate in the detail record. But the customer number is not appropriate as it should be the same for all of the detail items of a single order – it depends on order number but not on the sequence number. Solution: remove the customer number to the order header.

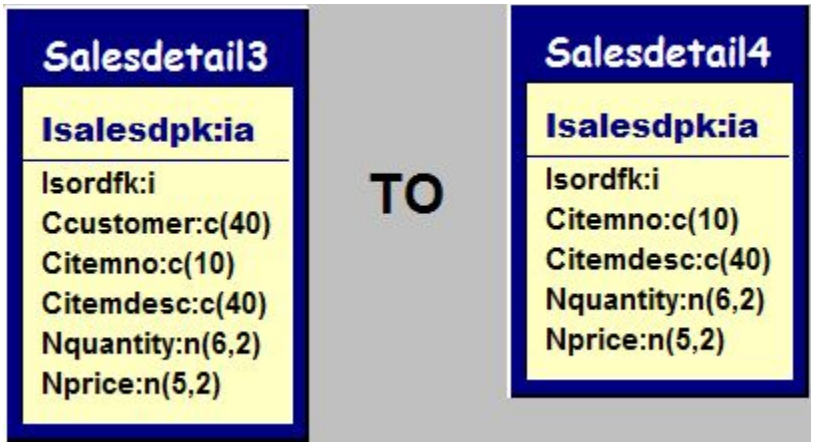

Figure 4: The customer identification should not be in the sales detail table, since it is the same for every detail of an order, it should be in the sales order header table.

Third normal form: Column values depend on the key and not on other column values.

Example: Order detail with ItemNo, Description, Price, Quantity. However, description should not change and is bound to ItemNo, not to the primary key, so it should be eliminated from the order detail record.

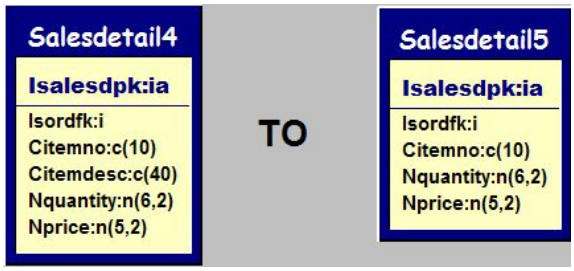

Figure 5: The item description is redundant, dependent on the item number and not the primary key. Therefore, it should be removed from the sales detail table to an item table.

Fourth normal form: (also known as Boyce-Codd normal form) Takes the rules of third normal form and applies them to all candidate keys.

Example: In a Sales Order Detail Table, there's a sequential line number that sorts the detail lines for a business requirement. The Order number is also included for support of another system. Together, they form a candidate key (another key that uniquely identifies each row and could be a candidate in the contest for primary key). If the customer number was included in the sales order detail, you can see that it is dependent on the order number field, but not the detail line field, pointing out that there is either a problem in the key selection or in the inclusion of this field in the table.

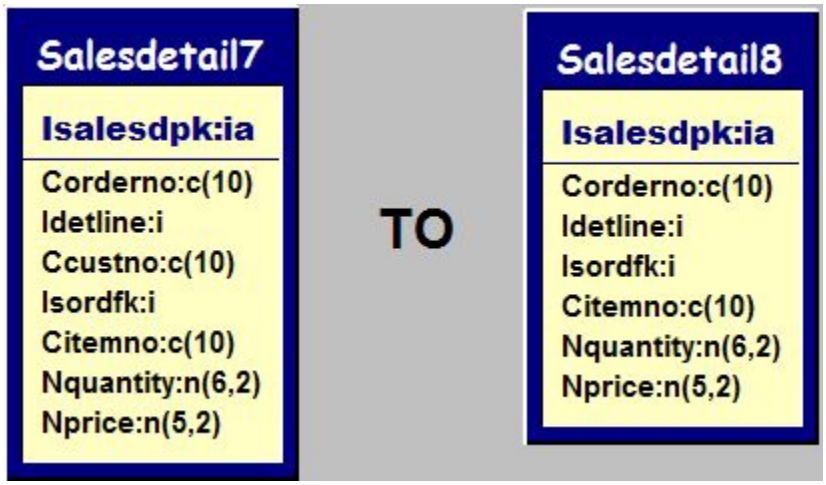

Figure 6: Boyce-Codd Normal Form requires 3rd Normal form for all fields based on the primary key and on any valid candidate keys. Order number and detail line form a natural primary key, but customer number depends only on the order number, not the line, and should be migrated to the order header.

So, what does all this normalization do for us?

Once the data has been normalized, it will be easier to manipulate, since each element of the data is represented in only one place in the data model. If the normalization step was

skipped, duplication of data would mean that repetitious code would need to be written to update, delete, and insert data into multiple places in the model. If you catch yourself writing code like that, check the data model for denormalization.

Relational Integrity Issues

Unlike the real world, we have the ability to make rules in the database that there can be no child record orphaned without a parent. We may also decide that the deletion of a parent record will result in the deletion of all related records, and that records can be entered only if their lookup values are valid. Different database engines enforce these rules in different ways. Some use **declarative referential integrity** (DRI) where the referential constraints are declared as part of the table definition. Visual FoxPro, like many other database engines uses a different technique of trigger-based referential integrity, where Insert, Update and Delete triggers fire to perform the appropriate rules enforcement. Visual FoxPro includes an RI Builder, accessible inside the data design surfaces, and generates RI code as stored procedures within the database container. There were and are issues with this builder; check out the Microsoft Knowledge Base and popular FoxPro forums for a list of issues and work-arounds.

Check out Stephen Sawyer's code at http://www.stephensawyer.com/ for an alternative to the Microsoft supplied RI Builder.

Non-data-bearing primary keys

One of the more common debates in data analysis is whether to use the “natural” primary key within the data or whether to create an artificial key whose sole purpose is to be the unique identifier for the record. There are valid arguments for both sides, and there are extreme cases where one side of the other has a clear advantage. In the middle, though, the answer comes down to compromise and choosing the standard that works for you.

The arguments against non-data-bearing primary keys include: bloated diskspace and bandwidth, potential to run out of keys, and the hassle of generating keys (pre-VFP8).

The use of non-data-bearing primary keys eliminate multiple-field keys, which are harder to manipulate. It also eliminates the need to ever change a primary key, since it has no meaning in the system. This, in turn, eliminates an entire class of relational integrity triggers.

My preference is to use the non-data-bearing primary keys, auto-incrementing integers when available, on every table.

Active flags vs. deleting records

Deleting a record removes information from the system. Deletion should only be allowed in the situations where no transactions were actually performed using that information. Fox example, if a record was inadvertently added twice, but no work done on that record, deleting the record doesn't change the data significantly. In other cases, the record should be flagged as inactive.

Example: Fred the Salesman leaves. Fred's Salesman record should be flagged as inactive. Clients assigned Fred as their salesman should be re-assigned. Sales that Fred

performed should continue to carry the PK of that salesman. Commission and payroll processing needs to support inactive flag, if applicable.

Deleting data can change history, so unless the system is a current snapshot and all past history is journaled in such a way as to support deletions, add flags to entities likely to be "logically deleted" so they can be flagged as inactive instead.

Naming Conventions

Visual FoxPro suggests several naming conventions in its Help file; search for "Naming Convention." Here are my general suggestions:

- Single character datatype, per the Help file (d for Data, y for Currency, etc.)
- Field names limited to ten characters for ease of use with free tables, temporary cursors and matching index names
- Keys end in "PK or "FK" and have the table name abbreviated, i.e., iCustPK is the customer primary key in the Customer table and iCustFK is the customer foreign key in another table.
- Avoid the use of underscores as they consume precious space.
- Avoid plurals both for space consideration and for consistency. You would expect the Client table to contain more than one Client, right? So Clients is pretty obviously just another letter consumed. Some gurus argue that tables should always be plural. Whichever technique you choose, be consistent!
- Consistency is the single most important guideline. If a number is "Num" in one table, "No" in another and "Nr" in a third, you will never get them right. Choose one and establish it as the standards.
- Write down all of your conventions (something called "documentation").

Naming conventions have powerful and beneficial effects in a system. When different prefixes are used for variables (the 'l' for local, 'g' for global, etc.), the chances of a field name and a variable being confused is very unlikely, saving you from some of the more difficult-to-debug problems. When applied consistently, they make the code easier to read and understand.

Using Tools

Man is a tool-using Animal. Nowhere do you find him without tools; without tools he is nothing, with tools he is all.
Thomas Carlyle

I think tools are an absolute necessity for a good data design. The tools do not need to be fancy, have gigabytes of RAM or cost thousands of dollars, although there are some neat toys out there that meet those criteria.

There are several purposes to data design:

1. Completely analyze the problem to ensure you have covered the entire problem space.
2. Creating a document to help you and others remember and understand the system.

Tools also let you interact with the data model, or generate scripts to effect changes. The xCase product, used to generate the data diagram illustrations here, has a fantastically flexible engine that can do all of the above, and let you interact, two-way between the data model and the data on disk. In addition, the xCase metadata is stored in dbf format, where it's relatively easy to query and perform global changes. I have no financial interest in the product, but I do encourage you to check it out.

Quick Facts

Product: xCase, www.xcase.com

Version: 7.4, April 2004

Price: $399 (Fox) to $799 (Professional) in several different packages

License: commercial, 1 per user

Features: two-way design interaction & scripting, color printing, data migration, target data server switching, much more

Documents produced by case tools not only make inexpensive wall art, but can become very useful tools for developers to reference on a daily basis. Since the tools can manipulate data structures, it is a great idea to only allow changes to the data model through the case tool, ensure the documentation is always up to date.

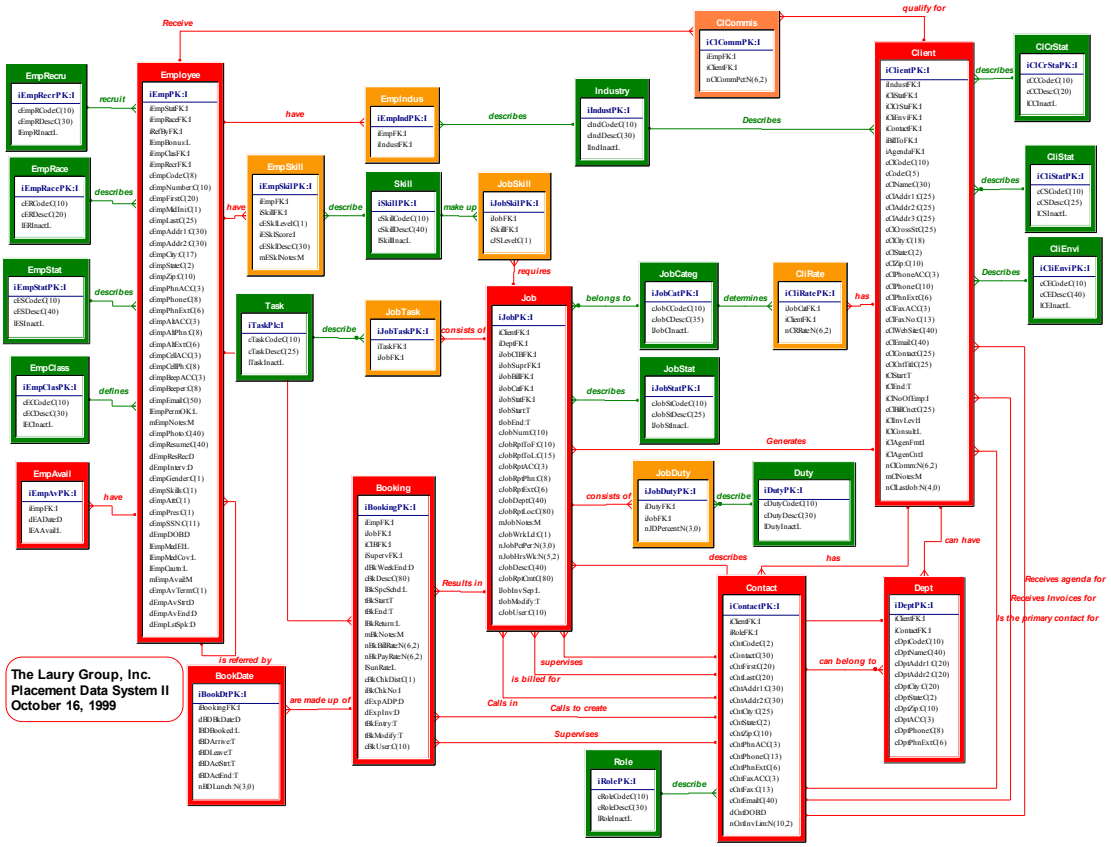

Figure 7: An Entity-Relationship diagram for a moderately complex application. For ease of presentation, the primary tables are in red, many-to-many tables in orange, and lookup tables in green.

Real-World Problems

Fun as all the theory it, putting it into practice can be challenging. Let's discuss, though

not necessarily solve, a couple of classic real-world problems.

People, Places and Things

Contact information for people and companies can be very difficult to model. There are many people in the typical business transaction: yourself, your employees, your customer, their employees, the delivery people, your lawyer, your accountant, an outside inspector, third-party vendors, etc. One common design decision is to lump every human being in the system into a single table, People, and then map them into all the places they fit in the data model. While it is the appropriate decision for some systems, for others, it can be a fatal error. From the logical analysis standpoint, a vendor is a vendor and customer is a customer. If you later decide to blend them into a shared table, that is a later optimization, and not a wise first step.

Another important step in the logical analysis phase is to declare the rules of the system in as simple a set as possible. It's important to understand that programmers and analysts try hard to work out all of the “edge” and “corner” cases to ensure they understand the valid one-to-many and many-to-many relationships in a system. However, it is easy to get carried away with this “Yeah, but what it...” game and turn it into a game of “Can You Top This,” whereby everything is related to everything in a many to many relationship.

Let's consider a phone contact management system. For simplicity, all contacts are the same; – their classification and grouping is outside the scope of the example. Each contact may have one or more phone numbers. It is desired to designate a “primary” or “main” phone number. Here's one possible design, first laid out as declarative sentences, and then diagrammed.

- There are two main entities: contacts and phone numbers.
- A contact can have many phone numbers.
- One phone number could be designated as the main number.
- Different phone numbers should be differentiated by “types.”
- There may be zero or many phone numbers for each type.
- For this model, contacts stand alone; they do not share phone numbers with other contacts.

There are several different solutions to this problem, and each have advantages and disadvantages, benefits and downsides. You'll want to examine and exercise the model carefully to ensure it meets your needs. One of the interesting questions I found when presenting this model was the question of whether an individual contact should be allowed multiple phone numbers of the same type. While the datqa model can surely be developed to support it, the user interface can become intricate, difficult and confusing. Microsoft Outlook solves this problem by having a fixed set of phone types, including “Home” and “Home2,” “Work” and “Work2,” as well as “Other,” and restricting entry to one phone number per type. A similar solution is worth considering.

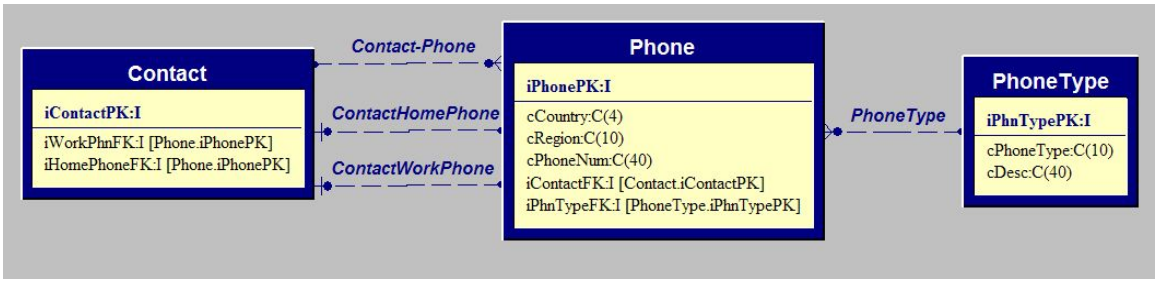

Figure 8: One proposed solution to the contacts problem has a one-to-may relation from contact to phone, and two phone foreign keys stored in the contact table in a one-to-one relationship.

Documents as data

Another interesting problem to consider occurs in a whole series of applications that can be generalized as supporting a document of information. That document could be a facilities inspection, a written test, or an assembly process. Common elements are used, assembled following some sort of formula, and results are recorded. Over time, not only do components change, but the recipes to assemble them change as well. Developing this model can be an involved process. The trick is to balance the needs of the system with the cost of adding complexity. A more complex model is more difficult to implement and maintain, and it can introduce unintentionally "legal" data entries that don't have an analogue in the real world.

For example, an inspection is the intersection of a facility with a scheduled series of tests with results that might either be finite (True/False), a range (0-100 ppb), a text blob (narrative) or a binary blob (actual test result output). An individual inspection takes place with a version of set of tests (the "current" one, hopefully). When later reviewing historical material, you can use the preserved version information to determine if a particular test was included in the then-current regimen to decide if the test results should be included or excluded from the historical analysis.

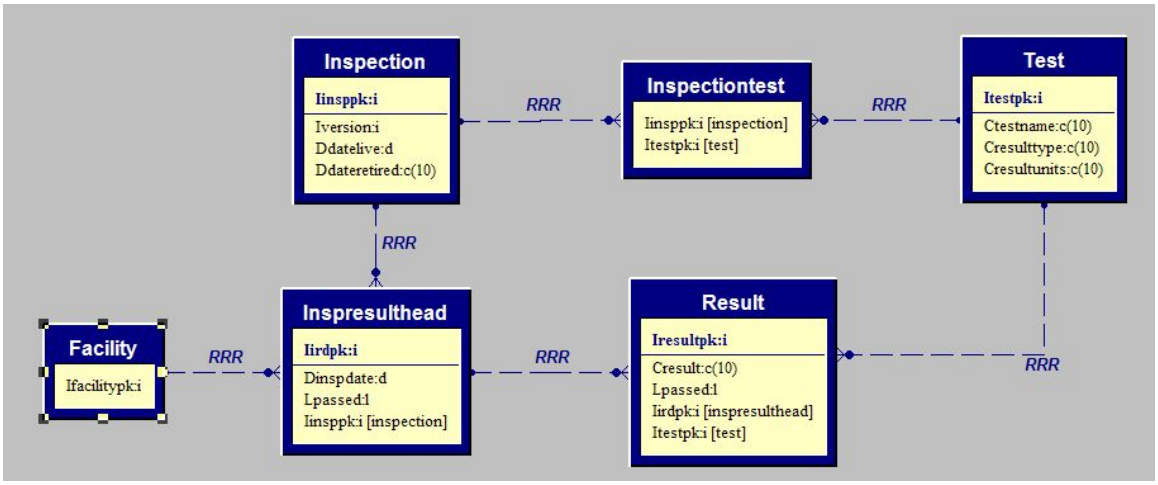

Figure 9: An example of a document model: inspections performed on facilities are a set of tests (1:M) and their results.

Conclusion

Data analysis and design is a critical early step in the successful development of an nal

application. Taking care in the early phases to ensure that design is well thought-out - well normalized, standardized and appropriate for the problem – has a multiplier effect down the line in the success of developing, implementing and maintaining an application.

About the Author

Ted Roche develops data-centric applications for Web, client-server and LAN use. He is a principal in Ted Roche & Associates, LLC, http://www.tedroche.com, where he offers consulting, training and mentoring as well as software development services. Ted is author of Essential SourceSafe, co-author of the award-winning Hacker's Guide to Visual FoxPro 6.0, and a contributor to five other FoxPro books.

To Learn More:

Akins, Marcia, Andy Kramek, Rick Schummer, *1001 Things You Wanted to Know About Visual FoxPro*, 2000, Hentzenwerke, ISBN 0-9655093-3-8, especially Chapter 7, "Working With Data"

Akins, Marcia, Andy Kramek, KitBox column in *FoxTalk* magazine: December 2003: "Addressing, the issues" covers the name and address modeling issue, and July 2004: "Let 'er rip" discusses hierarchical organization data models.

Booth, Jim and Sawyer, Stephen, *Effective Techniques for Application Development with Visual FoxPro 6.0*, 1998, Hentzenwerke, ISBN 0-96550-937-0, especially Appendix Two: Relational Database Design.

Fleming, Candace C. and von Halle, Barbara, *Handbook of Relational Database Design*, 1989, Addison-Wesley, ISBN 0-201-11434-8

Hernandez, Michael J., *Database Design for Mere Mortals*, 1997, A-W Developers Press, ISBN 0-201-69471-9