# Ten Things to Avoid in a Data Model

www.ERwin.com

Modelsoft Consulting Corporation
Dr. Michael Blaha
blaha@computer.org
www.modelsoftcorp.com

# Abstract

A data model helps developers think deeply about a database and cope with large size and complexity. However, given their high level of abstraction, data models can be difficult to construct and prone to errors. This article provides a checklist of some mistakes you should be sure to avoid, thereby improving the quality of your data models.

# Introduction

A model is an abstraction of some aspect of a problem [1]. A data model is a model that describes how data is represented and accessed, usually for a database. The construction of a data model is one of the more difficult tasks of software engineering and is often pivotal to the success or failure of a project.

Many factors determine the effectiveness of a data model [4] [6] — too many to fully address here. Instead we discuss pitfalls that developers can overlook. By avoiding these pitfalls, you can improve your models.

Our focus is not on detailed modeling constructs such as keys, data types, nullability, and referential integrity. Rather we emphasize the deeper essential content of a model.

## Ten Things to Avoid in Data Model

Things to Avoid – Modeling Strategy

1. Vague Purpose
2. Literal Modeling
3. Large Size
4. Speculative Content
5. Lack of Clarity

Things to Avoid – Modeling Details

6. Reckless Violation of Normal Forms
7. Needless Redundancy
8. Parallel Attributes
9. Symmetric Relationships
10. Anonymous Fields

# Things to Avoid – Modeling Strategy

Here are five high-level items to avoid.

## 1 Vague Purpose

Don't build a model without understanding the business rationale. The purpose for a model dictates the level of detail (just entities and relationships, fully attributed, with data types and full constraints). The purpose also dictates the level of polish, the degree of completeness, and the amount of time that can be justified.

There are different kinds of data models. A detailed application model is needed to guide application development. A rough application model suffices for conveying purchase requirements to a vendor. An enterprise model reaches across applications enabling integration, data warehouses, and other broad activities.

## 2 Literal Modeling

As a modeler, your job is not to do what the customer literally says — you won't get credit for following instructions if the project fails. Rather your role is to solve the problem that the customer is imperfectly describing. Therefore, you must pay attention to the hidden true requirements. You must interpret and abstract what the customer tells you.

We are sensitized to the trap of literal modeling by development with use cases. (A *use case* is a piece of customer-specified functionality for a system.) Many developers record use cases, while never thinking deeply about their content. Developers can fall into the trap of slavishly building exactly what the customer describes.

The alternative to literal modeling is thoughtful modeling with abstraction. There can be different ways of satisfying customer needs. In particular, you must recognize arbitrary business decisions that could easily change. Often you can avoid such vulnerability by

abstracting a model a bit.

For example, we encountered an application with separate tables for *IndividualContributor*, *Supervisor*, and *Manager*. *IndividualContributors* report to *Supervisors* and *Supervisors* report to *Managers*. This model is correct, but has problems. There is much commonality between the tables—for example, all three have phone numbers and addresses. The only difference is the reporting hierarchy.

Figure 1 shows an excerpt of the original model and an improved model that is more abstract. Instead of "hard coding" the management hierarchy, we can "soft code" it with a boss relationship. A person who has an *employeeType* of "IndividualContributor" reports to another person with an *employeeType* of "Supervisor." Similarly, a supervisor reports to a manager. A worker has an optional boss, so the reporting hierarchy eventually stops. The improved model is smaller and more flexible.

You can also raise abstraction is by thinking in terms of patterns [1]. A pattern is a model fragment that is profound and recurring. A *pattern* distills the knowledge of experts and provides a proven solution to a general problem. Patterns also boost productivity in building models.

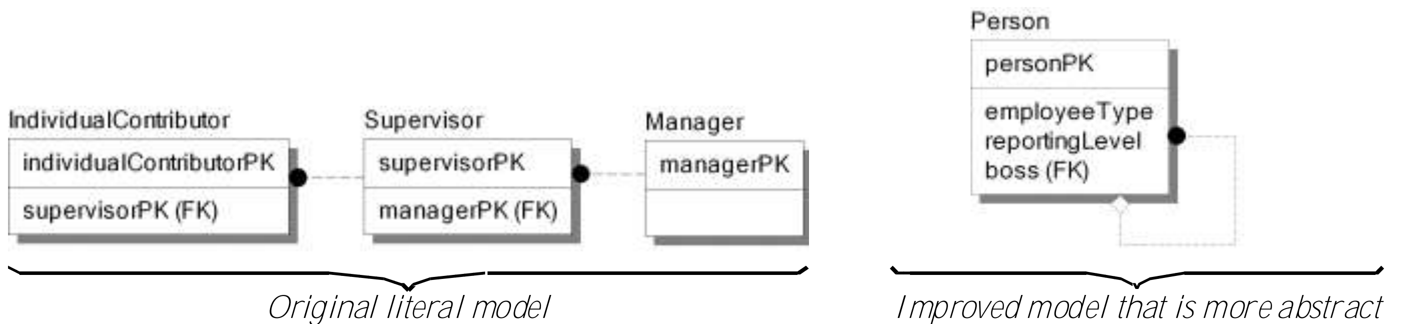



Figure 1 – Don't fall in the trap of literal modeling. Use abstraction to increase business flexibility.

Chapter 12 of [4] presents the Data Model Scorecard®. Scorecard item 5 (how well does the model leverage generic structures) corresponds to our warning about *literal modeling*.

## 3 Large Size

Over the years we have learned to avoid large models. As a general rule, you should try to limit a model to no more than 200 tables. The reason is that large models involve more work. You must ask yourself if the large size is really justified or if you can simplify the model through some well-placed abstraction. In our consulting work, we often reverse engineer existing databases. We rarely find a large model with a compelling justification.

Reference [2] illustrates the downsides of large models. For a data interchange project, we built an equipment data model that was 100 pages long. Equipment has a wide and deep taxonomy and we documented the details for the many different types. Figure 2 shows a very small excerpt. In retrospect we should have used an abstract model that defined types of equipment and their attributes with metadata.

When I build models, I now have an explicit step to reconsider the model's size. Is there extraneous content that I can remove? Can I shift the representation and make a model more concise and adaptable? How much work is it to develop the application for the model and is it worthwhile from a business perspective?

You should organize models with more than 50 tables into subject areas. A *subject area* is a group of elements (entity types, relationships, generalizations, and lesser subject areas) with a common theme. Subject areas help readers understand portions of a model at a time, rather than having to deal with the whole model at once.

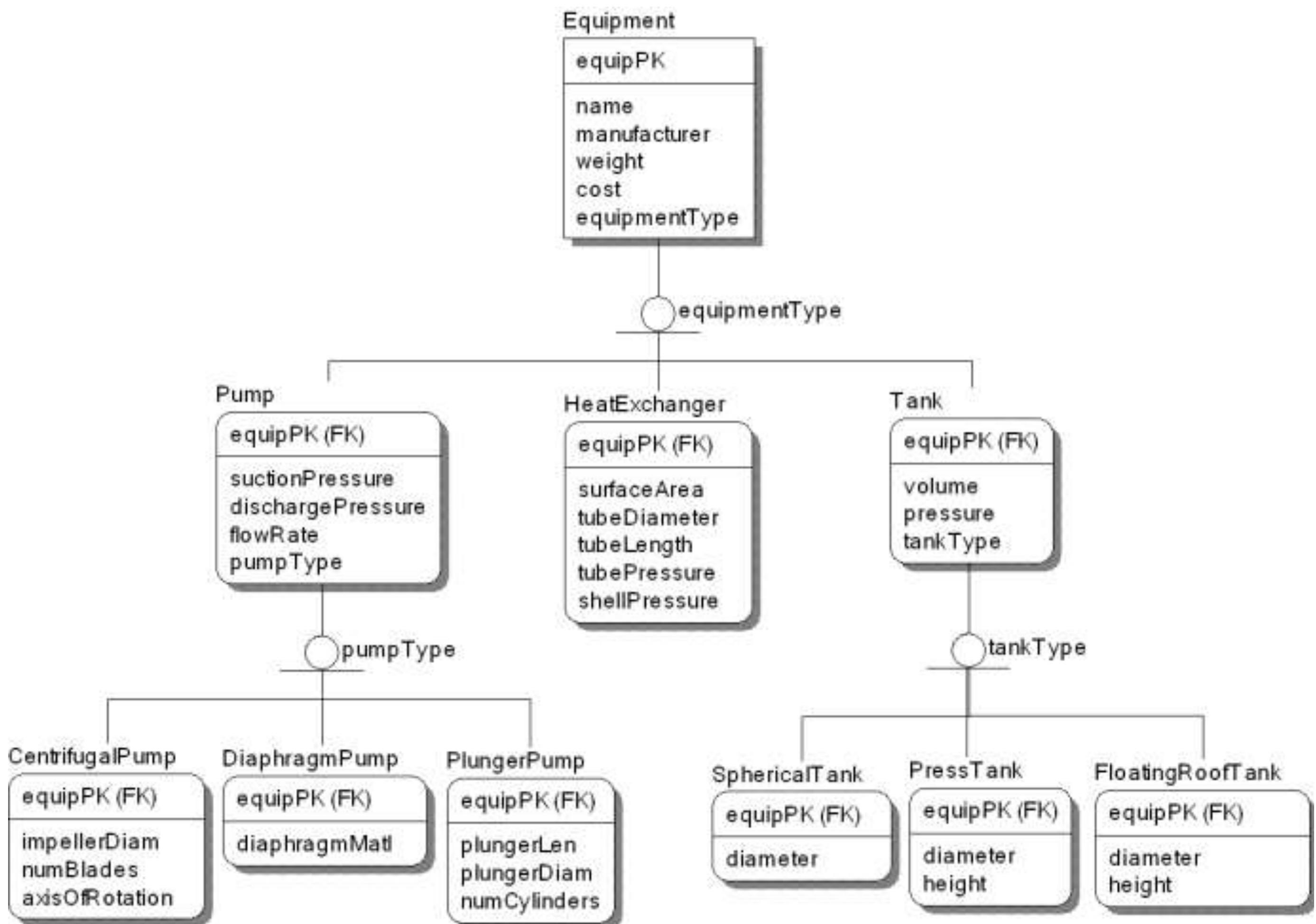



Figure 2 – Try to avoid large models. A shift in representation can often lead to a more concise and effective model.

## 4 Speculative Content

It is not a good idea to include content that is not needed now and "might be helpful" in the future. All this does is to make a model larger, increase development time, and raise cost. A model must fully address the requirements, but not greatly exceed them. At least 90 percent of a model should pertain to immediate needs. As much as 10 percent can anticipate future needs. Otherwise you run the risk of scope creep.

Aside from size and cost, another reason for limiting speculative content is that a quality data model should be intrinsically sound and readily extended. New requirements should cause additions, but little alteration. The addition of 10 percent to a model, should cause rework of no more than 10 percent of the original model. If you can readily extend a model, there is no point to adding content in advance of need. When you really need more content, the new requirements will be clearer than if you try to forecast them in advance.

Keep in mind that the purpose of modeling is to represent data relevant to your business objectives. Some people lose focus and model extraneous information. It can be reasonable to model a bit beyond your needs—after all, the exact scope of an application is seldom known up front and is partially a matter of negotiation. However, you do not want to reach way beyond application needs, because such a model is speculative and may never lead to something useful.

In [4] the Data Model Scorecard® item 3 (how complete is the model) corresponds to limit *speculative content*.

## 5 Lack of Clarity

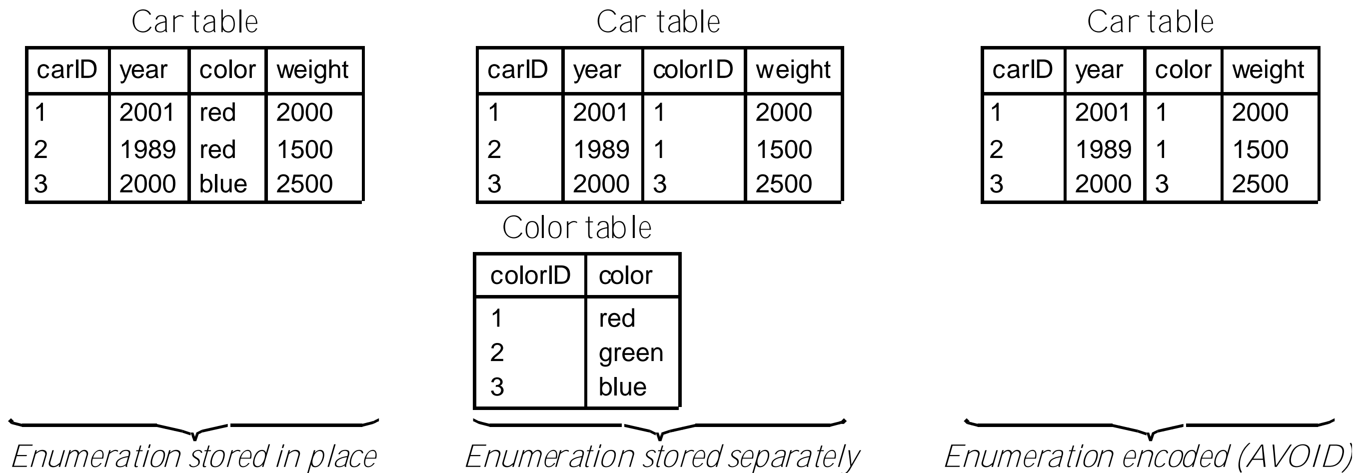
Normally a model should be free of obfuscation. (A model of a secure military application could be an exception.) A relational database is declarative and your model should adhere to the same spirit. So declare data in your models.

ERwin has a nice feature where you can define domains. A *domain* is the set of possible values for an attribute. You define the domains for your application and then assign them to the pertinent attributes. For example an application may have attributes *customerName*, *projectName*, and *productName* that can be assigned the *name* domain. Then they have a consistent data type and length that is defined in one place.

Enumerations are a special kind of domain. An *enumeration* is a domain that has a finite set of values. For example, a car may have a color that can be red, green, or blue. Do not define enumerations in programming code and store the encoding in the database. Instead declare enumerations in your database. Enumerations often arise in applications. They appear in user interfaces as pick lists. Business experts often mention enumerated values when describing an application.

As Figure 3 shows, enumerations may be declared or encoded. Declared enumerations are stored as a string in place or in a separate table (either is reasonable). Encoded enumerations are stored as a number which application code must translate (this is undesirable).

Domains and enumerations are just two aspects of declaring data. In addition, don't store data structures with a binary encoding. Don't use cryptic names. Don't use anonymous fields that application code must interpret.




| carID | year | color | weight |
|---|---|---|---|
| 1 | 2001 | red | 2000 |
| 2 | 1989 | red | 1500 |
| 3 | 2000 | blue | 2500 |

| carID | year | colorID | weight |
|---|---|---|---|
| 1 | 2001 | 1 | 2000 |
| 2 | 1989 | 1 | 1500 |
| 3 | 2000 | 3 | 2500 |

| colorID | color |
|---|---|
| 1 | red |
| 2 | green |
| 3 | blue |

| carID | year | color | weight |
|---|---|---|---|
| 1 | 2001 | 1 | 2000 |
| 2 | 1989 | 1 | 1500 |
| 3 | 2000 | 3 | 2500 |

Figure 3 – A model should have clarity. Declare data, such as enumerations, in your models.

# Things to Avoid – Modeling Details

Here are another five things to avoid, this time looking at models in detail.

Some of our detailed advice varies for operational and analytical applications. An *operational application* (such as an order entry system) concerns the routine operations of a business. Operational activities tend to be simple, access few records, and respond within seconds.

In contrast, *analytical applications* emphasize complex queries that read large quantities of data and enable organizations to make strategic decisions. Analytical applications execute against a data warehouse. Data warehouses are skewed towards reading and their structure does not enforce data quality — that is the job of the loading programs. Thus some of the detailed modeling anomalies are acceptable for data warehouse applications.

# 6 Reckless Violation of Normal Forms

A *normal form* is a guideline for relational database design that increases data consistency [3]. As tables satisfy higher levels of normal forms, they are less likely to store redundant or contradictory data. Normal forms are not fiat rules. Developers can violate them for good cause, such as to increase performance for a bottleneck, such as tables that are read and seldom updated. Such a relaxation is called *denormalization*. The important issue with normal forms is to violate them deliberately and only when necessary.

Figure 4 shows an example of a denormalized table and the normalized restatement. The contact position and contact phone depend on the contact name which in turn depends on the primary key of the table. Such indirect dependencies violate third normal form. Several customer records could have the same contact person as indicated by contact name. The problem with the current design is that the various references to a contact need not have the same position and phone. A separate contact table can enforce uniformity of position and phone.

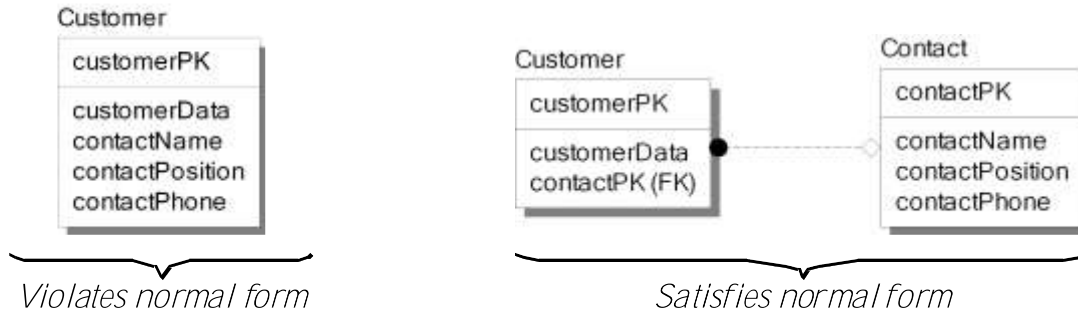



Figure 4 – Do not violate normal forms, except for analytical applications and performance bottlenecks.

Denormalization is only justified for operational applications when there is a major performance bottleneck. In contrast, denormalization is routine practice for analytical applications because it simplifies queries. The negative effects on data quality are unimportant as a data warehouse relies on the loading programs for data quality.

**Pay attention to large tables in operational applications as the large size may indicate denormalization. A 'large' table has** 30 attributes or more.

Denormalization is often caused by mixing entity types. So you should suspect any entity type that is difficult to define. An entity type should be simple and coherent. Otherwise, unless you have a situation where denormalization is justified, you should split a composite entity type into its fundamental constituents.

# 7 Needless Redundancy

Observing normal forms alone is not sufficient — the larger issue is avoiding needless redundancy. The violation of normal forms is undesirable precisely because it causes redundancy. Ideally a database should have a single recording of each data item.

Redundancy is a plague of information systems. Organizations are rife with applications that overlap in awkward and loosely controlled ways. In fact, one of the major justifications for data warehouses is having a reporting system that reconciles conflicting data [5].

**Redundant data is seldom helpful. Don't include redundant data in an attempt to compensate for a poorly conceived application**. Relational databases can deliver surprisingly good performance if an application is properly architected. It is difficult to keep redundant data consistent with the base data. There are issues of the latency of update, vulnerability to errors, and additional code to write.

Redundant data is more acceptable if you use built-in database features to keep redundant data consistent with base data. For example, materialized views can be helpful.

## 8 Parallel Attributes

Figure 5 illustrates parallel attributes. The left table has one record per project and stores individual project costs with parallel attributes.

**In contrast, the right tables treat cost symmetrically; nine records store a project's cost data.** *CostItem* is material, labor, or tax. *CostBasis* is estimate, reestimate, or actual. The two estimates are an artifact of the existing business process and could change. In the future, there could be one estimate or three estimates. Furthermore, there could be additional *CostItems*. The parameterized model preserves flexibility by not encoding such arbitrary business practices into the database structure.

Parallel attributes are acceptable for a data warehouse and are often used in dimensions to simplify queries. For operational applications, a few parallel attributes are inoffensive, but widespread use often indicates a poor model.

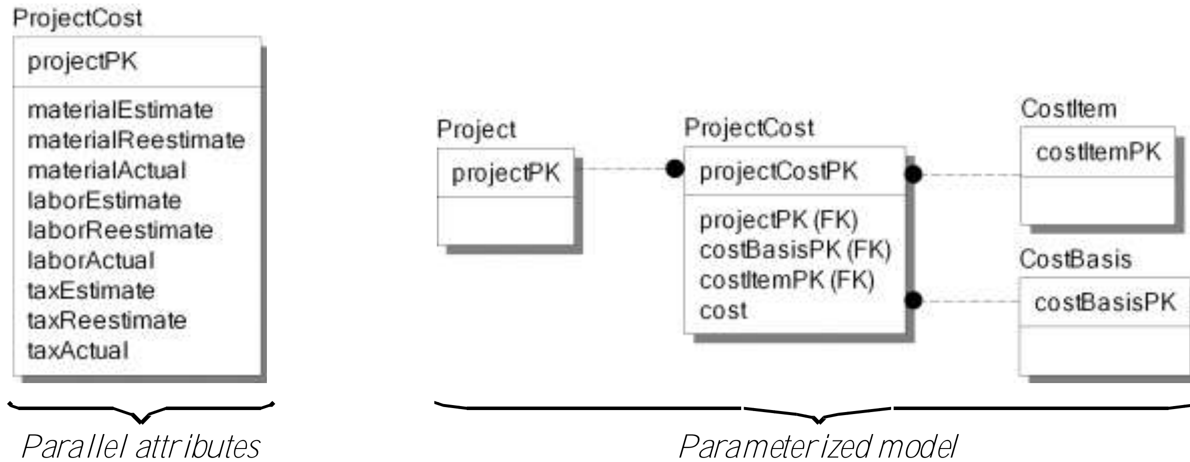



Figure 5 – Avoid parallel attributes for non-data-warehouse applications.

## 9 Symmetric Relationships

Symmetric relationships are troublesome for relational databases. Resolve them by promoting the relationship to an entity type. The improved model not only resolves the symmetry but is often more expressive.

In the left model of Figure 6 *RelatedContract* is a many-to-many relationship. If each pairing is entered once, it is not clear which contract should be first and which is second. If each pairing is entered twice, the amount of storage increases and any change requires double update. If more than two contracts are related, the situation is messier yet (for three contracts that are double stored: C1-C2, C2-C1, C1-C3, C3-C1, C2-C3, C3-C2). Furthermore, there is no requirement that the related contracts be different. None of this is desirable.

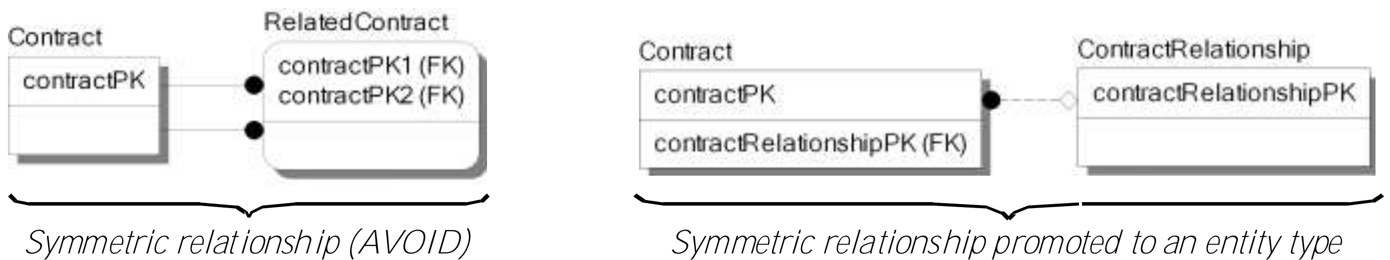



Figure 6 – Promote a symmetric relationship to an entity type in its own right.

The improved model (right of Figure 6) breaks the symmetry. To find related contracts traverse as follows: start with a *Contract*, find the possible *ContractRelationship*, then traverse back to *Contract* (excluding the initial contract) to obtain the related *Contracts*.

The revised model has further advantages. The coupling is no longer binary and can readily support three or more related contracts. The model could be extended to make *Contract* to *ContractRelationship* many-to-many with different relationship types. For example, one relationship type could be successor contracts (one contract replacing another). A second relationship type could be alternative contracts (several contracts being considered as alternatives for purchase).

## 10 Anonymous Fields

As much as possible, you should clearly describe the data being stored and not use anonymous fields.

Figure 7 shows an excerpt of a location table with anonymous fields. To find a city, you must search multiple fields. Worse yet, it could be difficult to distinguish the city of *Chicago* from *Chicago* street. Furthermore, you may need to parse a field to separate city, state, and postal code. It would be much better to put address information in distinct fields that are clearly named.

fragment of Location table

| locationAddress1 | locationAddress2 | locationAddress3 |
|---|---|---|
| 456 Chicago Street | Decatur, IL xxxxx | |
| 198 Broadway Dr. | Suite 201 | Chicago, IL xxxxx |
| 123 Main Street | Cairo, IL xxxxx | |
| Chicago, IL xxxxx | | |

Figure 7 – **Don't use the poor design practice of anonymous fields.**

# Conclusion

Data modeling is often the pivotal task in building a database application. A data model determines an application's data quality, extensibility, and performance — and ultimately whether or not the application has a chance at business success. Many factors determine the effectiveness of a data model. We present some of the possible pitfalls here, both from the perspective of modeling strategy as well as modeling detail. By avoiding these pitfalls, you can improve your data models.

# References

1. Michael Blaha. *Patterns of Data Modeling*. New York, NY: CRC Press, 2010.

2. Michael Blaha. Data store models are different than data interchange models. *ateM workshop, 10th IEEE Working Conference on Reverse Engineering*, November 2003, Victoria, BC.

3. Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th edition*. Reading, Massachusetts: Addison-Wesley, 2010.

4. Steve Hoberman. *Data Modeling Made Simple, 2nd edition*. Bradley Beach, NJ: Technics Publications, 2009.

5. W. H. Inmon. *Building the Data Warehouse, 4th edition*. New York; Wiley, 2005.

6. Graeme C. Simsion and Graham C. Witt. *Data Modeling Essentials, 3rd edition*. San Francisco, CA: Morgan Kaufmann, 2005.

# Biography

Since 1994 Dr. Michael Blaha has been a consultant and trainer in conceiving, architecting, modeling, designing, and tuning databases for dozens of organizations throughout the world. He has authored six U.S. patents, five widely used books, and many papers. His most recent book, *Patterns of Data Modeling*, was published in June 2010. Blaha received his doctorate from Washington University in St. Louis and is an alumnus of GE Global Research in Schenectady, NY. You can contact him at blaha@computer.org.