

Observed Idiosyncracies of Relational Database Designs

Michael R. Blaha* and William J. Premerlani**

*OMT Associates Inc., St. Louis, MO 63017 (blaha@acm.org)

**GE Corporate R&D, Schenectady, NY 12301 (premerlani@crd.ge.com)

Abstract

Several processes have been advanced in the literature for reverse engineering of relational databases. The inputs to these processes are relational tables and available contextual information. The output is a model of the underlying logical intent, apart from the implementation artifacts. Most of the existing processes for database reverse engineering are inadequate; they assume too high a quality of input information. The authors of these processes are skilled database designers and they are overly optimistic about the state-of-the-art, as practiced. This paper catalogs odd aspects of relational database designs that we have encountered over the past several years. Many of these database designs are from commercial software products.

1. Introduction

Most existing software engineering methodologies emphasize forward engineering—conceiving, analyzing, designing, and implementing an application from scratch. However, such a simplistic approach handicaps the software engineer. The reality is that much software is purchased and not developed. Furthermore, software that is to be developed often is not designed from scratch, but is intended to eclipse some existing system. Reverse engineering offers new capabilities that empower the software engineer.

We define reverse engineering as the extraction of logical intent from software implementation artifacts. This paper does not address the general reverse engineering problem, extraction of information from any software artifact. Rather our scope is more limited, extraction of information from a relational database. We agree with [2] that the most tractable approach for database applications is to first reverse engineer the database and then deal with the programming code.

Object-oriented models provide a natural language for facilitating the re-engineering process. An object-oriented

model can describe the existing software, the reverse-engineered semantic intent, and the forward-engineered new system. This paper adopts the Object Modeling Technique (OMT) notation for modeling data [4]. Graphical OMT models are intuitive yet provide a rigorous basis for specifying software. OMT concepts are similar to those in extended Entity-Relationship (ER) modeling.

In general, the mapping between object models and database schemas is many-to-many. Various optimizations and design decisions can be used to forward engineer an object model into a database schema. Similarly, when reverse engineering a database, alternate interpretations of the structure and data can yield different object models. Usually, there is no obvious, single correct answer for reverse engineering. Multiple interpretations can all yield plausible results.

A good way to begin reverse engineering is by entering the existing schema into a CASE tool. Associations will often be found in a degraded form such as relational database foreign keys. Inheritance must be implemented in a degraded manner for current relational database managers. The schema may then be gradually transformed to a logical model as underlying relationships are inferred.

There are many possible motives for reverse engineering of databases:

- **Migration between database paradigms.** One may want to migrate between database paradigms—from past hierarchical, network, and relational databases to modern relational and object-oriented databases.
- **Migration within database paradigm.** A more mundane task would be to migrate between different implementations of a database paradigm, for example from one vendor's relational database to another's relational database.
- **Documentation.** Reverse engineering can elucidate poorly documented existing software when the developers are no longer available for advice.
- **Tentative requirements.** Reverse engineering of existing software can yield tentative requirements for the new

replacement system. Reverse engineering ensures that the functionality of the existing system is not overlooked or forgotten.

- **Software assessment.** We have derived substantial benefit from reverse engineering of databases from vendor software. Reverse engineering provides an unusual source of insight for assessment. The quality of the database design is an indicator of the quality of the software as a whole. An understanding of the concepts supported by the underlying database schema allows one to better judge functionality claims.
- **Integration.** Reverse engineering facilitates integration of related legacy applications and purchased applications. A logical model of encompassed software is a prerequisite for integration.
- **Conversion of legacy data.** One must fully understand the logical correspondence between the old database and the new database before attempting to convert data.
- **Assessment of state-of-the-art.** From our perspective as methodologists, reverse engineering provides candid insight about the state of the database design art—as practiced.

An earlier paper [3] presented our process for reverse engineering of relational databases. Our process improved upon the literature by more flexibly coping with the design optimizations and unfortunate implementation decisions that are found in practice. [3] also described several unusual implementation techniques that we found. Since publication of the earlier paper we have had the opportunity to reverse engineer a number of additional databases and continue to be surprised by some database designs. This paper takes a much broader look at database design techniques.

This paper makes one major contribution to the literature. We systematically catalog the design quirks, optimizations, and flaws that we have encountered in practice with a number of relational databases (about twenty). We have not found any such compendium in the literature. Our list is admittedly incomplete. Nevertheless, we hope that the list will be informative and stimulate further improvement in reverse engineering technology, as a source of raw material to both methodologists and software tool developers. Reverse engineering is difficult to automate until we can more fully characterize the likely inputs.

2. Summary of OMT Notation

Figure 1 summarizes OMT constructs that are included in this paper. A class describes objects with common attributes, behavior, and semantic intent. A class is denoted by a rectangle. Attributes may be suppressed or displayed in the second portion of the class box. We underline attributes when we wish to indicate the primary key.

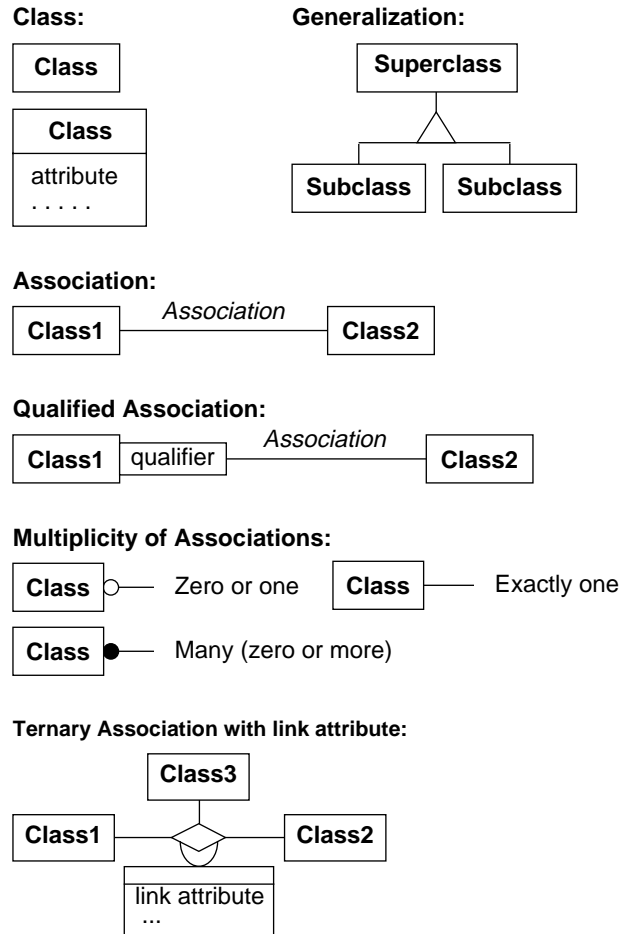


Figure 1 OMT syntax used in this paper

Generalization organizes classes by their similarities and differences and is denoted by a triangle. Simple generalization apportions superclass instances among the subclasses. OMT also supports several forms of multiple inheritance that are not shown in this paper.

An association relates instances of two or more classes and is indicated by a line. Multiplicity specifies how many instances of one class may relate to an instance of an associated class. An association may be qualified in which case the multiplicity is further refined by the qualifier attribute. For example a directory has many files but the combination of directory and file name corresponds to one file. A diamond denotes a ternary association, an association between three classes. The instances of an association may be described with attributes that we call link attributes. A box attached to the association by a loop denotes a link attribute.

3. Catalog of Idiosyncracies

We have encountered numerous idiosyncracies during reverse engineering of relational databases. Some of the id-

iosyncracies are proper design techniques, but complicate the task of reverse engineering. Other idiosyncracies are mistakes by the database designer that are desirable to remedy.

3.1 Implementation of classes

Typically each class maps to a table, but a class may map to multiple tables or multiple classes may map to one table (Section 3.5).

- **Fractured objects.** A class may be partitioned into multiple tables—horizontally, vertically, or both. For example, one table may contain high-level attributes and another table may contain detailed attributes (vertical partitioning). A class may also be partitioned horizontally: Two tables have the same schema with some objects in one table and some objects in another.

3.2 Primary keys

Ordinarily every table should have a primary key. However, exceptions do occur, such as tables with temporary data or tables for which the performance overhead cannot be tolerated. We have encountered several kinds of primary key flaws.

- **Missing primary key without cause.** Some applications are just sloppy. Other applications enforce primary keys with custom code and do not rely upon the database manager. We have seen applications that consist of multiple closely related programs where some programs enforce primary keys and some do not. For such applications the underlying database is vulnerable to corruption.
- **Null primary key attributes.** Some relational database managers require that one define a unique index to enforce a primary key. Indexed attributes are permitted to be null, unless not null is specified for each of the attributes. This violates the definition of a primary key; attributes in a primary key may not be null.
- **Extraneous primary key attributes.** By definition a primary key must also be minimal; no attribute can be discarded from a primary key without destroying uniqueness. We have found primary keys with suspicious attributes; no data records require the extra attributes and the attributes do not seem semantically justified. The reverse engineer must regard all primary key declarations with suspicion.
- **Primary keys that are almost but not quite unique.** In one application there were two classes in the schema that we thought should participate in a one-to-one association. However, there appeared to be an extra, unmatched record in one of the tables. Upon closer examination of the data in the table with the extra record, we discovered that the extra record is apparently used by the ID generating algorithm as a place holder for the next record in

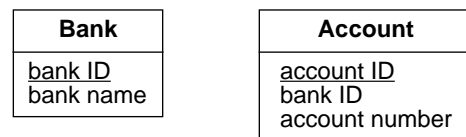
the table. A single-instance class was being combined with another class, breaking a strict one-to-one association. [3]

3.3 Approaches to identity

Even when tables do have a primary key, different realizations may still be chosen. In object-oriented software each object can be identified and distinguished apart from the attribute values that it may happen to have. Many relational database managers provide functionality helpful to realizing object identity. For example, Oracle supports the notion of a sequence; Microsoft Access has the counter data type. Relational databases allow one to implement the object-oriented ideal of artificial identity or to define identity in terms of combinations of attribute values.

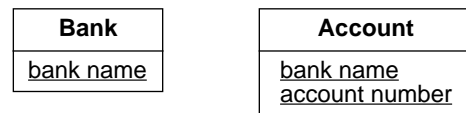
Figure 2 shows relational tables for three different approaches to identity. All three schema can be reverse engineered to the same logical model.

Reverse engineering input: Artificial identity

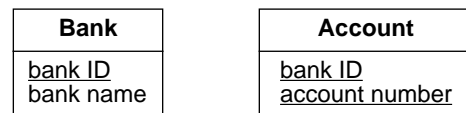


{Candidate key of Bank is: bank name.}
{Candidate key of Account is: bank ID + account number.}

Reverse engineering input: Value-based identity



Reverse engineering input: Hybrid identity



{Candidate key of Bank is: bank name.}

Reverse engineering output: Logical intent

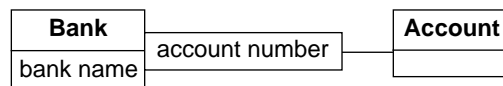


Figure 2 Various approaches to identity

- **Artificial identity.** Each object table (shown in Figure 2) has an object identifier as primary key. Association tables (not shown in Figure 2) have a primary key consisting of the identifiers of the related objects.

- **Value-based identity.** The primary key of each object consists of some combination of application attributes. Some primary keys may become lengthy, as attributes are incorporated from foreign keys of related tables. (See cascaded qualified associations in this section.)
- **Hybrid identity.** One may use artificial identity and value-based identity in the same schema. In the third segment of Figure 2 *Bank* has artificial identity and *Account* has identity derived from a reference to a bank combined with an account number.
- **Cascaded qualified associations.** Cascaded qualified associations are common. An accumulation of qualifiers denote increasingly specific objects. In Figure 3 a city is identified by the combination of a country name, state name, and city name. With value-based identity the primary key of city could be country name + state name + city name. The “1” in the *Earth* class is OMT syntax that denotes a singleton class, a class with a single instance. Singleton classes often initiate qualification cascades

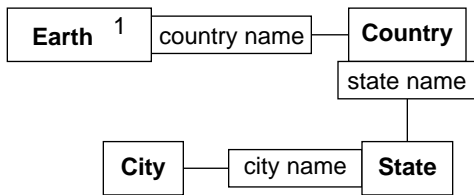


Figure 3 Cascaded qualified associations

- **Propagated identity.** When two classes are related by a one-to-one association, the primary key of one class may be used for the other class as shown in Figure 4. Usually, this is not a good idea; it contravenes the spirit of object-oriented development as there is a ‘leakage’ of information from one class to the other.

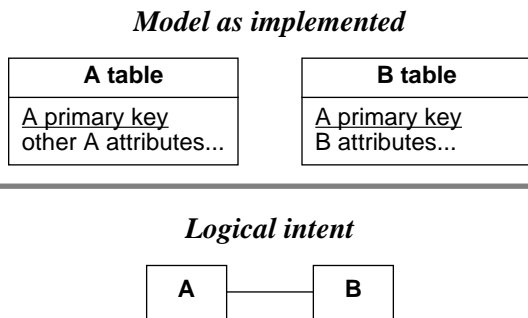


Figure 4 Propagated identity

3.4 Implementation of associations

Many-to-many associations are typically implemented with a dedicated association table (also called intersection table). One-to-many and one-to-one associations may also

be implemented with an association table or by burying a foreign key. For a one-to-many association the foreign key is buried in the ‘many’ class pointing to the ‘one’ class. For a one-to-one association the foreign key may be buried in either class. For all these schemes, the foreign key attribute name may or may not match that of the referenced primary key. Several other permutations are occasionally used.

- **Double-buried associations.** On several occasions we have found an association buried in both participating classes as shown in Figure 5. This can be a reasonable design technique as the association can be rapidly searched in both directions. However, this construct complicates reverse engineering. At first glance double-buried associations look like two separate associations. Data analysis can detect redundancy between the dual pointers, but semantic understanding is required to resolve this situation.

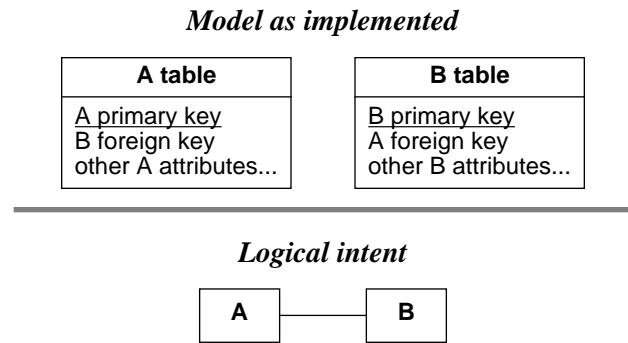
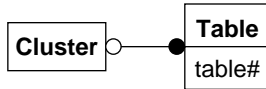


Figure 5 Double buried association

- **Optional qualified association.** Figure 6 shows an optional qualified association from [3]. A *Cluster* contains many *Tables*. A *Table* may belong to at most one *Cluster*. The combination of a *Cluster* and a *table#* yields a specific *Table*. This association was implemented by burying *cluster_id* as a foreign key in *Table*. Because of the optional membership in a cluster, the foreign key can be null, and the combination of *cluster_id* and *table#* is not a candidate key of *Table*. Therefore it is difficult to detect this qualified association; in fact, both diagrams would produce exactly the same schema. In this case, we believe the qualified association is a more accurate representation of the designer’s intent
- **Alternate qualifier.** The example in Figure 7 is taken from the data dictionary of a relational database. [3] A *Column* derives its identity from a *Table* plus a qualifier, either *column name* or *column number*. In fact, the association pairs column names with column numbers. At first, when we encountered this one, we thought there were two separate associations.
- **Ternary and n-ary associations.** We seldom use ternary and n-ary associations when we model a new applica-

Model as implemented

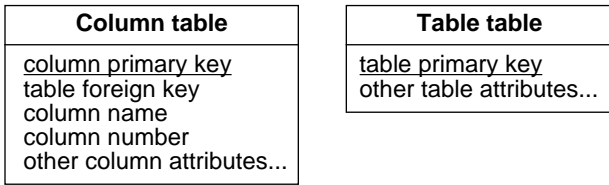


Logical intent



Figure 6 Optional qualified association

Model as implemented



{Candidate key of Column table is:
table foreign key + column name,
table foreign key + column number.}

Logical intent

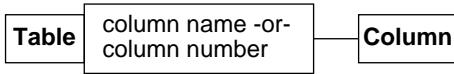


Figure 7 Alternate qualifiers

tion from scratch. However, when we reverse engineer databases, we frequently find ternary associations; the confluence of primary keys from three or more classes are required to identify the records in an association table. We regard reverse engineered ternary and n-ary associations as highly suspicious. We believe that most of these associations are an artifact of implementation and the uncertainties of reverse engineering rather than fundamental information. Where possible, ternary and n-ary associations should be restated as binary associations.

3.5 Denormalization

Logical constructs are sometimes combined in a manner that violates normal forms. On occasion, it is acceptable to violate normal forms, but such decisions should be willful and for good cause.

- **Multi-class tables.** A table may combine two or more classes with the intervening association(s); such a table may or may not satisfy normal forms. The table in Figure 8 satisfies third normal form, as the coordinates can be perceived as directly depending on the particular rectangle. In contrast the table in Figure 9 violates second normal form as the plant manager depends on only part of the primary key.

mal form as the plant manager depends on only part of the primary key.

Model as implemented: Table in 3NF

Rectangle
rectangle name top left x coordinate top left y coordinate bottom right x coordinate bottom right y coordinate

Logical intent

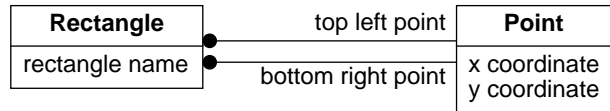


Figure 8 Multi-class table in third normal form

Populated table violating 2NF

Equipment				
plant name	equipment name	plant manager	manufacturer	address
ethylene	final cooler	Jim Smit	Z Best	1247 Third
ethylene	feed heater	Jim Smit	Z Best	1247 Third
styrene	feed pump	Art Gunn	XYZ	14 Pine Dr.
styrene	feed heater	Art Gunn	Z Best	1247 Third

Logical intent

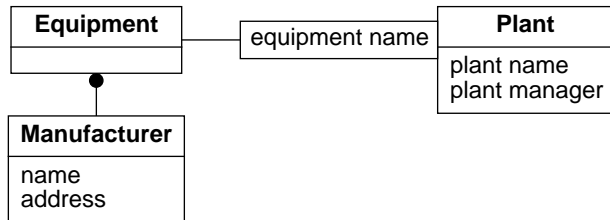


Figure 9 Multi-class table violating second normal form

- **Degradation of inheritance.** Current relational databases lack semantic support for inheritance. One strategy is to define separate superclass and subclass tables. With this strategy referential integrity can enforce the correspondence between objects in a subclass and those in the superclass. However relational databases cannot express the generalization partition, that each object in a superclass is further described in a single subclass. One can also push attributes down to subclasses, but this is ver-

bose and contrary to the spirit of normal forms. Normal forms strive to store each fact in a single location. By pushing attributes down to the subclasses, the schema is stored in multiple places. A third option is to push attributes up to the superclass, but this violates second normal form.

- **Parallel attributes for a repeating group.** Technically this satisfies normal forms, but is still ugly. Information should be stored directly and simply. It can be difficult to search parallel attributes and enforce integrity constraints. In Figure 10 parallel attributes are used to record monthly sales. A better design would be to store multiple records in a table with attributes *year*, *month name*, and *monthly sales*. We have also seen parallel attributes used to implement an x-to-many association.

Model as implemented

Corporate sales
year
January
February
March
April
May
June
July
August
September
October
November
December

Logical intent

Corporate sales
<u>year</u>
<u>month name</u>
monthly sales

Figure 10 Parallel attributes for a repeating group

- **Sequence number to resolve x-to-many association.** In one database design a sequence number had been introduced solely to resolve a one-to-many association, so that it could be buried on the ‘one’ side as shown in Figure 11. Of course this badly violates second normal form as the A attributes are replicated in multiple records. We talked to the designer and discovered that he was simply confused about how to properly implement one-to-many associations with tables. Surprising to us, on several occasions we have met database designers who are perplexed about how to implement one-to-many and many-to-many associations. Often these persons are skilled programmers and novice database designers.

Model as implemented

A table	B table
<u>B primary key</u> sequence number A attributes...	<u>B primary key</u> other B attributes...

Logical intent



Figure 11 Sequence number for one-to-many association

3.6 Referential integrity

Ideally referential integrity constraints (SQL foreign key clauses) should be enforced by the database manager rather than custom coded for each application. In the past this was not always possible, as some database managers did not support referential integrity. Sometimes bonafide performance concerns intrude. And sometimes database designers are misguided, overlooking the benefits of a uniform declarative mechanism that assures referential integrity.

- **Informal linkage between tables.** Tables may be informally linked by human inspection, rather than formally linked via referential integrity. For example, in Figure 12 the company name is embedded in the job name. Each job pertains to a customer company; a company may have many jobs. The linkage *may* be easy for a human to detect, but can be difficult for a machine. First of all, the linkage between the columns is not noted in the data dictionary. However, even if the linkage is known, the precise computational relationship varies and is arbitrary. In Figure 12 company name is a substring of job name, but there are case differences (Acme vs. acme) and punctuation differences (AT&T vs. A.T.&T.). This kind of chaos is common in poorly controlled systems.
- **Mismatched referential integrity domains.** Domains may mismatch where referential integrity logically applies. In Figure 13 the *person number* in the association table (data type of number) refers to the *person number* in the *Person* table (data type of string). The number data type elides leading zeroes while the string data type preserves leading zeroes. Aside from being misleading, such careless assignment of data types to domains complicates joins and searching of the database.
- **Foreign key reference to candidate key.** Figure 14 illustrates the situation where a foreign key refers to a candidate key rather than to the primary key. This does not violate normal forms, but we would argue as does [1]

Populated tables

Company

<u>company name</u>	address	phone number
Simplex	100 S Main	555-1234
Acme	111 Broadway	555-3141
AT&T	201 Potomac	555-1812

Job

<u>job name</u>	estimated start date	estimated duration	estimated cost
Simplex new PC	Jan 1, 1990	1 month	\$5,000
Acme PC repair	Jul 3, 1986	2 weeks	\$200
A.T.&T. LAN upgrade	Feb 9, 1987	3 months	\$20,000
acme disk repair	Oct 5, 1986	2 weeks	\$50

Logical intent

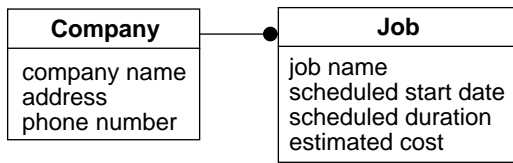


Figure 12 Informal linkage between tables

that referential integrity via candidate keys is a bad design practice.

- **Illegal foreign key.** The data dictionary for a relational database manager contained an interesting optimization of a qualified association. [3] As shown in Figure 15, the association between *Constraint* and *Constraint_column* is qualified by *position*. As an apparent optimization, the association is implemented in two different ways, depending on whether or not there is more than one column in the constraint. If a constraint contains exactly one column, *position* is set to null. Because *position* can be null, it cannot be part of a unique index. Consequently, we had some difficulty at first in detecting the qualifier.

3.7 Null values

Relational database schema frequently are lax with regard to not null constraints. Often a schema will allow nulls due to an inability to set a value within the scope of a single transaction. Furthermore, there is some argument about the desirability of using null values in the first place. [1] We have discovered different workarounds for the problems of null values.

Populated tables

Person

<u>person Number</u>	person Name
123456789	Jim Smith
005683201	Jane Smith
135246798	George Lee

Project

<u>project Number</u>	projectName
1234	new market- ing system
2345	new finan- cial system

Person-Project-Week table

<u>person Number</u>	<u>project Number</u>	<u>week Number</u>	hours
5683201	1234	1	20
5683201	2345	1	20
135246798	1234	1	40

Logical intent

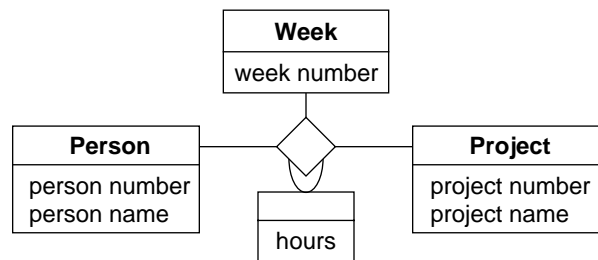


Figure 13 Mismatched referential integrity domains

- **Permissiveness.** A not null constraint that logically applies may not always be specified. One must carefully interpret schema.
- **Null representation.** We have seen various strategies: use null, use default value, or use a strange value that does not arise in the application.

3.8 Ambiguity

Ambiguity frequently arises with schema to be reverse engineered. One may receive a copy of the database schema without any accompanying explanation. Often business processes evolve while the software remains unchanged; software users compensate by giving attributes and tables a different meaning than the original intent. Often the names in database schema are obtuse, complicating the task of reverse engineering.

- **Anonymous names for tables and attributes.** In one application we found the table shown in Figure 16. We are not sure what these attributes mean. Our best guess is

Populated tables

Vehicle		
<u>vehicleNumber</u>	year	make
8573028573	1977	Ford Granada
7366625548	1989	Mazda 323

Legal entity		
<u>number</u>	name	address
849572089	Simplex	100 S Main
123456789	Jim Smith	4 Pine Drive
005683201	Jane Smith	4 Pine Drive

Vehicle-Owner	
<u>vehicleNumber</u>	ownerName
8573028573	Simplex
7366625548	Jim Smith
7366625548	Jane Smith

Logical intent

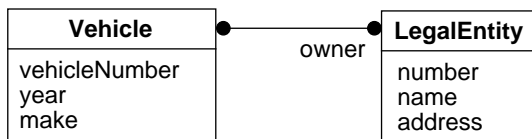


Figure 14 Referential integrity via candidate key

Logical intent

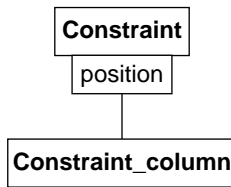


Figure 15 Illegal foreign key

that this table is being used as an intermediary for populating a spreadsheet.

- **User-defined attributes.** A user-defined attribute is a placeholder that allows the end user to add descriptive information not addressed by the basic software. Many commercial software packages include user-defined attributes. User-defined attributes complicate reverse engineering; the meaning of data in user defined columns of tables may not be obvious.
- **Mislabeled attributes.** Often software is constructed with an original purpose that fits business needs. Since

Model as implemented

.....
d100
d200
d300
d400
d500
d600
d705
d710
d720
d725
d740
d750
d775
d790
.....
h200
h300
h400
h705
h710
h720
h725
h740
h750
h790
.....

Figure 16 Ambiguous attributes in a table

software is difficult to extend and maintain, a business may redefine fields to store different data without updating the schema. With user-defined attributes, one lacks a description of the meaning of the data. Mislabeled attributes are much worse, as one is given a completely misleading description of the data.

- **Overloaded attributes.** More than one attribute may be stored in a column of a table. Sometimes the different values that are stored may be differentiated by a switch in another column. Other times the different values are distinguished by their format or contextual knowledge of the application.

3.9 Domains

Relational database theory supports the notion of a domain, but most database manager products do not. This lack of support for domains complicates reverse engineering.

- **Lack of enforcement for enumeration domains.** We have frequently encountered enumerations in our database designs and in those of others. An enumeration may become corrupted when multiple applications access a database and some applications lack enforcement.
- **Encoded enumerations without deciphering.** Unfortunately it is common for enumeration values to be encoded

ed in an inscrutable manner that completely obscures the semantic meaning.

- **Inconsistent formats.** Inconsistent formats complicate queries for data analysis during reverse engineering. For example, a phone number may be simply a series of digits or include a dash to separate the area code from the remainder of the number.

3.10 Redundancy

Redundancy is often introduced into relational database schema for performance reasons. The problem is that the data dictionary does not distinguish between fundamental information and derived information. Redundant information can be quite confusing during reverse engineering.

- **Intermediate tables.** Intermediate tables can cache computed results and provide opportunities for tuning in the middle of complicated database computations. For example, a database may store detailed accounting information for the hours each person spends on each project in a week. A roll-up table may store the accumulated hours charged to date by project.
- **Redundant attributes.** It is not uncommon to find two attributes in a table that store the same information, but in different forms.
- **Derived associations.** Derived associations are common. For example, the association between *Conglomerate* and *Department* may be derived from an association between *Department* and *Company* combined with an association between *Company* and *Conglomerate*.
- **Duplicate records.** We have found tables with completely duplicate records. This is sloppy and could easily be averted with use of primary keys.
- **Contradictory data.** Worse yet, we have seen tables with contradictory data. On one occasion, this occurred through merger of two separate databases and lack of integrity enforcement in the database schema.

3.11 Software engineering issues

During reverse engineering one must also consider software engineering issues.

- **Multiple design styles.** A schema that we studied for a commercial product had about 500 tables. Different design styles were evident—different approaches to implementing associations (buried or as separate tables). Also there was some variation in naming conventions.
- **Accumulation of historical changes.** Similarly with the same product mentioned above we could see that the design style for newly defined tables was different than that for older tables. The change of design style over time may have been a consequence of changing personnel.

- **Relic tables.** Some databases have obsolete tables from past releases of the software. These can be quite confusing to the reverse engineer. One should be suspicious of tables with zero records.
- **Multiple versions of tables.** Some products keep multiple copies of tables that are slightly different to facilitate migration of data from a past release of the software to the current release.

3.12 Complex situations

- **Generic tables.** A generic class is a class that combines data and metadata. In Figure 17 *Window parameter* is a generic class that pairs the name of a window default attribute with its corresponding value. Generic classes allow one to accommodate classes and attributes that may not be known as the application is written. Generic tables are not difficult to reverse engineer.

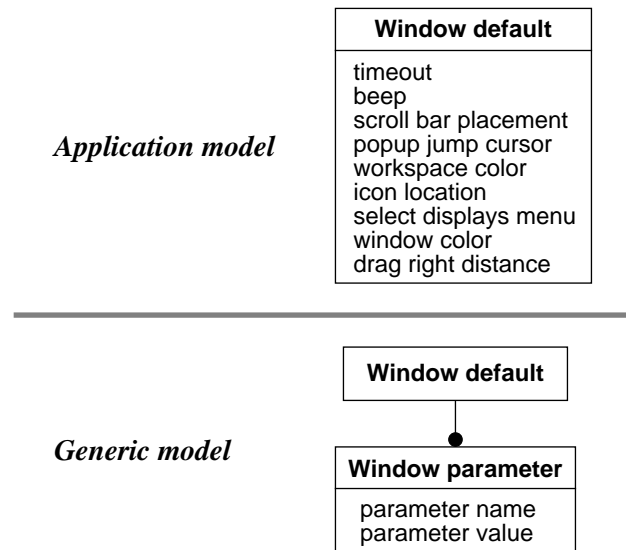


Figure 17 Generic tables

- **Attribute metadata.** Attribute metadata is occasionally found. For example, many scientific applications involve units of measure, such as inches, meters, seconds, and joules. One may wish to note the source of data, whether the values are obtained from persons, the literature, calculations, estimates, or other sources. One may require the time of the last update for each attribute value.
- **Non-object tables.** Functions, constraints, and enumerations may be represented as tables. Such tables are not directly represented in the reverse engineered object model.

3.13 Strange situations

We have found two downright strange situations during database reverse engineering. We are surprised that designers would even consider these techniques. We shudder to think about what other possibilities may be lurking.

- **Parallel attributes with a continuation flag.** [3] In Figure 18 the primary key of the table is (*puid*, *pseq*). Attributes *pvalu_0* through *pvalu_9* are foreign keys on table *User*. One record in *List of receivers* points to as many as ten users. When all ten array elements in a record are filled, another record is created with the same *puid* and the next available value for *pseq*. All associations that are implemented in this manner use the same attribute names for the elements of the fixed length array. We had the opportunity to talk to the database developers. We discovered that they were proficient C++ programmers who were inexperienced at relational database design and made their best attempt. This mistake was found in a commercial product

Model as implemented

List of receivers
puid:CHAR(15)
pseq:NUMBER(22)
pvalu_0:CHAR(15)
pvalu_1:CHAR(15)
pvalu_2:CHAR(15)
pvalu_3:CHAR(15)
pvalu_4:CHAR(15)
pvalu_5:CHAR(15)
pvalu_6:CHAR(15)
pvalu_7:CHAR(15)
pvalu_8:CHAR(15)
pvalu_9:CHAR(15)

Figure 18 Parallel attributes implementation of an association

- **Hierarchical database within a relational database.** This is another really odd one and was also found in a commercial product. The product supports a hierarchical database within a relational database. The hierarchical database is a fixed tree with a depth of three. As we began analyzing this one, we were perplexed. We understood the problem domain but could find only a smattering of associations. We could not understand how so much information could be missing. Then we suddenly

realized that some of the relationships may be disguised. We cross checked the schema with the hierarchical screen layout and discovered the missing associations. We confirmed our understanding with the software vendor. Apparently the vendor created the product a number of years ago with a proprietary hierarchical database on a personal computer. Then the vendor migrated to client-server technology and devised an isomorphic hierarchical database under a relational database to correspond to their proprietary client database manager. Figure 19 shows the structure for level 1, 2, and 3 tables.

Model as implemented

Level 1 table	Level 2 table	Level 3 table
_internid	_internid	_internid
_chgdate	_chgdate	_chgdate
_chgtime	_chgtime	_chgtime
_stamp1	_stamp2	_stamp2
.....	_stamp3
	

Figure 19 Hierarchical database within a relational database

4. Conclusions

This paper systematically lists the design quirks, optimizations, and flaws that we have encountered in practice with a number of relational databases. We hope this will be useful experimental data for methodologists and tool makers.

References

- [1] CJ Date. *Relational Database: Selected Writings*. Addison-Wesley, Reading, Massachusetts, 1986.
- [2] JL Hainaut, M Chandelon, C Tonneau, and M Joris. Contribution to a theory of database reverse engineering. *International Conference on Software Engineering, Workshop on Reverse Engineering*, May 1993, Baltimore, Maryland.
- [3] William J Premerlani and Michael R Blaha. An approach for reverse engineering of relational databases. *International Conference on Software Engineering, First Working Conference on Reverse Engineering*, May 1993, Baltimore, Maryland.
- [4] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.