A Copper Bullet for Software Quality Improvement

Most software engineers agree that software quality improvement comes not from a silver bullet but from a combination of strategies. One of these copper bullets is to reverse-engineer a vendor's database as part of an evaluation to determine overall quality before buying the product.

Michael Blaha
Modelsoft Consulting Corp.

As Fred Brooks described in his famous "no silver bullet" paper, there is no single action the computing community can take to radically improve software quality. There are, however, "copper bullets"—lesser steps that improve quality over time. One such copper bullet is the notion of software engineering, the practice of thinking carefully before immersing yourself in the minutia of coding. Judiciously applied, software engineering should improve quality.

In theory, that makes sense, but for better or worse, software is becoming larger and more complex, which makes the benefits of software engineering less noticeable. As the software community faces an unprecedented number of project failures, researchers must continue the hunt for new copper bullets to offset the complexity.

A quality improvement strategy that the community has largely ignored is to use database reverse engineering to measure the quality of software that a company is looking to buy. Companies now routinely assess vendor software on the basis of cost, functionality, user interface, and vendor stability, but none of these dimensions addresses the software's intrinsic quality. Database quality, on the other hand, could be a litmus test for overall quality. If a product has a flawed database, it is likely to have other quality issues, such as messy programming. In contrast, the quality evident in a sound database is likely to be present in the software's other parts.

Over the past 11 years, my colleagues and I have been evaluating software using database quality as the basis for product grading. We have found that reverse-engineering a database can help a company deeply understand the associated product. Moreover, the time to do the evaluation (sometimes only a few person-weeks) is trivial compared to the millions it can cost to buy and deploy the application.

The benefits of this copper bullet are enormous. As reverse engineering pressures vendors to improve their offerings, vendors upgrade their software development practices to survive the scrutiny. Success then depends less on marketing prowess and more on technical merit.

Proficient vendors receive more notice, can negotiate more attractive prices, and can look forward to increased sales. Inferior vendors receive less revenue and are eventually forced from the marketplace as companies flock to their more proficient competitors. Openness is encouraged, since reverse engineering makes it plain what models and database designs vendors actually offer, regardless of what the vendor is willing to publish.

Thus, as more companies practice database reverse engineering, aggregate vendor quality should improve, benefiting the entire software community.

The suggestions offered here extend only to database assessment, not to the more general problem of reverse engineering to assess code. I have found that this is hardly a limitation, however, since large companies tend to buy mostly information systems, which

0018-9162/04/$20.00 © 2004 IEEE

Table 1. Grading scale used in reverse-engineering databases.

Grade	Explanation	Design flaws	Model flaws
A	Clean	• No significant flaws	• Style is reasonable and uniform
B	Structural flaws, but they don't affect the application (can be repaired without much disruption)	• Data types and lengths not uniformly assigned • Not-null constraints not used to enforce required fields • Unique keys and enumerations not defined • Columns have cryptic names like *Cell123*	• Anonymous fields that application code must interpret
C	Major flaws that affect the application (bugs, low performance, difficult maintenance)	• Undefined primary keys • Propagated identity • Haphazard indexing • Foreign-key data type mismatches primary key • Parallel foreign keys	• Needless complexity • Excessive inheritance • Specific modeling errors
D	Severe flaws that compromise the application	• Much unnecessary, redundant data • Extensive binary data (compiled programming language data structures), subverting the declaration of data • Gross denormalization • Dangling foreign-key references	• Lack of crisp conceptualization • Many arbitrary restrictions
F	Appalling (the application won't run properly or runs only because of brute-force programming)	• Gross design errors	• Deep conceptual errors

location_address_1	location_address_2	location_address_3
456 Chicago Street	Decatur, IL xxxxx	
198 Broadway Dr.	Suite 201	Chicago, IL xxxxx
123 Main Street	Cairo, IL xxxxx	
Chicago, IL xxxxx		

Figure 1. Database design flaw—anonymous fields.

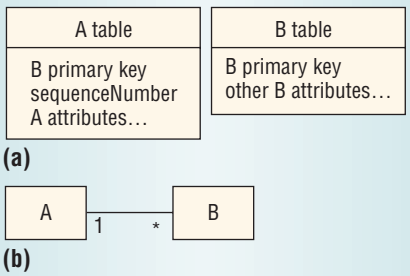


Figure 2. Database design flaw—using a sequence number for a one-to-many relationship: (a) tables as implemented and (b) the logical intent.

are built around a database. I've also found that vendors are more willing to provide their database structure than programming code, which makes database reengineering a more realistic improvement strategy.

WHY REVERSE ENGINEERING?

Forward engineering is the process of building software. Development flows forward from product conception, through analysis and design, and then finally to implementation. Most developers view this flow as more iterative than sequential, given the endless reviews and feedback loops. Reverse engineering begins at the end, with the application code, and works backward to deduce the requirements that spawned the software.[1]

Reverse engineering is certainly not new. Indeed, savvy developers use it to study existing applications and salvage useful ideas, data, and code. To my knowledge, however, few software engineers have applied reverse engineering to vendor assessment. I believe this is an oversight and that reverse engineering of vendor products should be a routine aspect of all software evaluations.[2]

CONNECTING DATABASE AND PRODUCT QUALITY

When we began evaluating databases in 1992, we decided to adopt the grading system in Table 1 to summarize the results of reverse engineering. We have found that business leaders readily understand the meaning of the grade, realize that we have the supporting technical details, and appreciate being allowed to study the details at their leisure. As the table shows, we assess the quality of both the database design and the conceptual model that underlies the database using A, B, C, D, or F, with A being the best grade and F the worst.

The first time we performed database reverse engineering, it was as an experiment. We were studying a vendor product and were perplexed by our experiences. The vendor had a great marketing story and clearly understood the business requirements. The

company was both large and credible, so we expected high-quality software. When we encountered a number of problems with the product, we decided to look at the database and discovered that its poor quality was at the root of the problems.

This experience prompted us to look at additional databases, after which we decided that database reverse engineering was not an odd technology, but something we should routinely perform. We started keeping records of our experiences and have amassed 11 years of data in a grading table (available in its entirety at www.modelsoftcorp.com). I believe our results represent broad practice, given that a different team prepared each database. We evaluated databases only from developers we did not advise as part of our consultant work, and we included databases only for applications that a vendor actually completed.

The case studies include both vendor assessments and in-house reengineering and where possible indicate if the application succeeded or failed, with "success" defined as actual use. Of course, by definition, the case studies are biased toward success, given that we (as customers) did not see the products companies scrapped.

QUALITY PROBLEMS

Our assessments revealed many applications with flawed databases. More important, database quality has improved little in 11 years. At best, I can give only a flavor of the problems we encountered, but we did see many recurring design flaws. Several databases had anonymous address fields, for example, which satisfies database design theory, but is sloppy nonetheless.

Consider the data in Figure 1. To find a city, you must search multiple fields. Worse yet, it could be difficult to distinguish Chicago the city from Chicago the street. Furthermore, you might need to parse a field to separate city, state, and postal code. A better design would put address data in distinct fields that are clearly named.

Some database designs used a sequence number to resolve a one-to-many relationship so that it could be buried on the "one" side, as in Figure 2. The sequence number is a completely meaningless field, making it difficult to find data in the A table.

Another common flaw was overloaded foreign keys. In the tables of Figure 3, each address can be linked either to a person or a company (targetID) as indicated by switch.

Several databases had propagated identity, as in Figure 4. From reading the database textbooks, you might think that propagated identity is fine, but

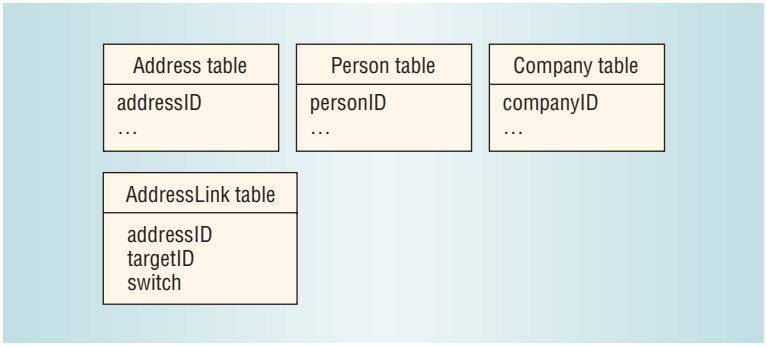


Figure 3. Database design flaw—overloaded foreign keys.

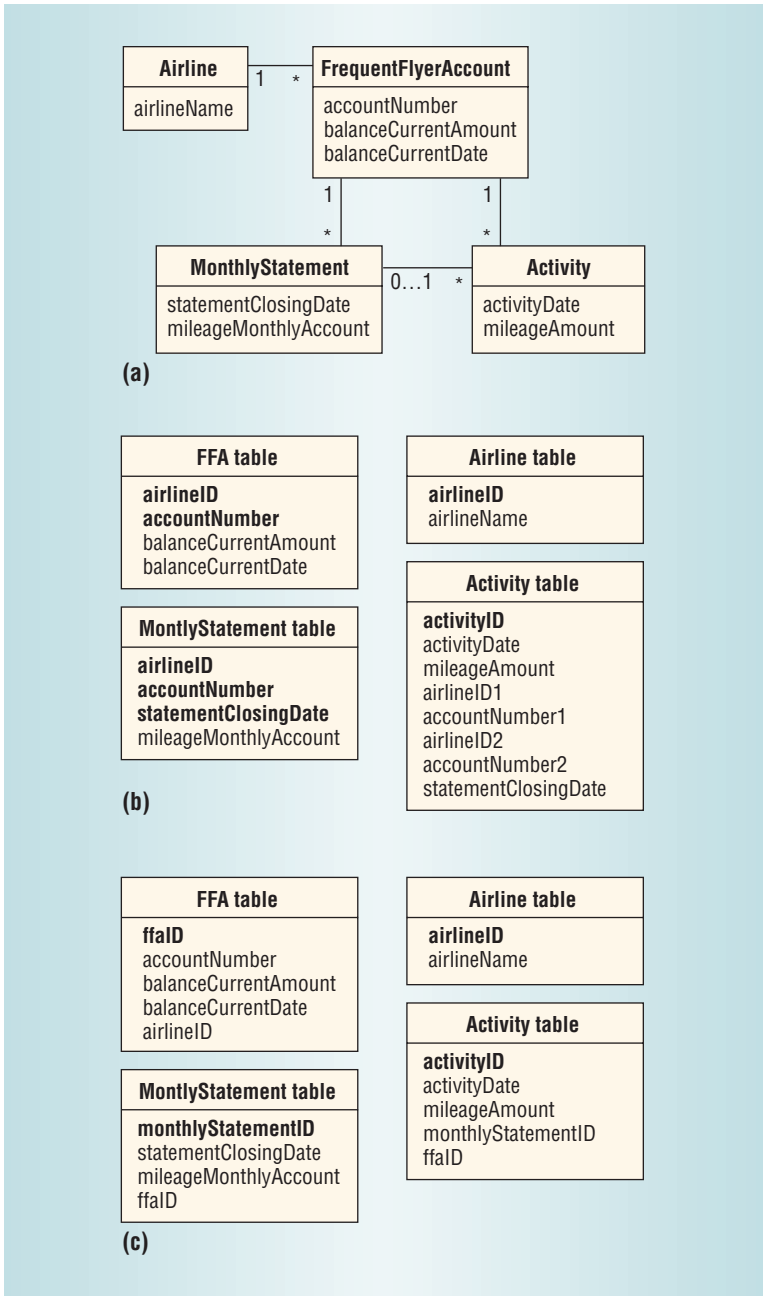


Figure 4. Database design flaw—propagated identity: (a) UML model; (b) implementation with propagated identity, which causes two sources to propagate a field; and (c) a better implementation with existence-based identity.

Company table

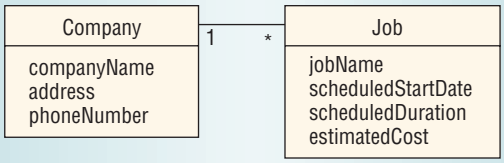
companyName	address	phoneNumber
Simplex	100 S. Main	555-1234
Acme	111 Broadway	555-3141
AT&T	201 Potomac	555-1812

(a)

Job table

jobName	estimated StartDate	estimated Duration	estimated StartDate
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

(b)



(c)

Figure 5. Database design flaw—informally linking tables: (a) company table, (b) job table, and (c) logical intent.

from a modeling perspective, it is clearly an inferior approach. Propagated identity leads to multiattribute primary keys and can give rise to the situation in which a field is propagated from two sources, but must be duplicated to enforce referential integrity. In the Activity table in Figure 4b, for example, airlineID1 and airlineID2 really represent the same field, but the design requires two separate fields—two copies—to satisfy referential integrity. Thus, AirlineID1 + accountNumber1 refers to the FFA table. AirlineID2 + accountNumber2 + statementClosingDate refers to the MonthlyStatement table. Unfortunately, with SQL, a field cannot refer to two tables, so a single airline field cannot refer to both FFA and MonthlyStatement without causing mathematical ambiguity. Figure 4c shows a correct implementation with existence-based identity, which eliminates the need for duplication.

One database informally linked tables by human inspection, rather than formally linking them via referential integrity. In Figure 5a and 5b, the company name is embedded in the job name. Each job pertains to a customer company; a company can have many jobs. The link may be easy for a human to detect, but it can be difficult for a machine. First, the link between the columns is not declared in the database. However, even if the link is known, the precise computational relationship varies and is arbitrary. In Figure 5b, 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.).

INTERPRETING RESULTS

The data we collected in our reverse-engineering case studies proved to be a useful indicator of historical software quality. Table 2 summarizes the grades converted to points: A = 4.0, B = 3.0, C = 2.0, D = 1.0, and F = 0. Although there is some scatter in the data, the table shows clearly that database designs and models have improved little over the past 11 years. My colleagues and I suspect that the increased use of database design tools could explain the modest improvement in database design quality. Even so, the average database design and conceptual model remain mediocre.

The data essentially confirms our conclusion from anecdotal observation that modeling shows little improvement. Models baffle many developers, who do not appreciate the leverage that modeling can provide in building applications. Roger Box and Michael Whitelaw were quite accurate in observing that abstraction is the most difficult aspect of modeling.[3]

We believe that the root cause for the lack of improvement is that universities are not teaching students how to model software. Many universities teach the syntax of modeling, but they don't teach the art and thought processes. We presume that most professors are not teaching modeling because they don't know how to do it themselves.

We also found a correlation between vendor applications and software developed in-house. As Table 3 shows, the two have comparable quality, which means that software houses are not necessarily more professional than IT departments within corporations, as some might expect.

REVERSE ENGINEERING AND ETHICS

Many articles tend to give reverse engineering a sinister image, implying that developers typically use it to re-create a product.[4] In all our case studies, we made it clear that this was *not* our goal. Besides being unethical, reimplementation is usually uneconomical. Instead, we assured vendors that our focus was to assess the software's merit, to get past hidden assumptions and the sales claims,

and to gain a deeper understanding of the product so that we could better communicate with the vendor and use the software more effectively. In short, when we assess products, we are merely trying to determine what the vendor is selling.

When we reverse-engineer a product, we openly ask vendors for their database structure and tell them why we want it. If they refuse, we tell them we will penalize them in the evaluation. We performed many of these reverse-engineering case studies under commission from large companies. Large companies emphatically do not want to devote their resources to re-create a product. Commercial software is important to them, but it is incidental to their primary business. Otherwise, they would be writing their own software, not purchasing it. The industrial mentality is to outsource work that is not a core competency, so rewriting commercial software is one of the last things these companies want to do. In light of that, most vendors acquiesce and settle for a nondisclosure agreement to protect them from competitors. We encourage our clients to agree to reasonable nondisclosure terms and to make it clear to the vendor that there is no intent to compromise its technology or reveal its secrets to competitors.

Some vendors might find reverse engineering threatening, but it should worry only the inept. Superb vendors should welcome the process because it makes their excellence visible in a much more credible way than words or an impressive sales ad.

Database quality is undeniably a good indicator of application quality. I have found that business leaders welcome the insights gained from database reverse engineering and use them to make more informed decisions about purchasing an application. More important, the benefits have the potential to ripple into the entire computing community. If vendors improve, in-house software development will follow suit. The same personnel, over time, move between vendors and in-house staff, and demand from vendors and customer companies will pressure universities to teach students better.

At this time, only pockets of people are assessing vendor software with database reverse engineering, and I would like to see the practice spread. If everyone would reward excellent vendors and penalize sloppy ones, we could improve overall software quality, not just in databases, but eventually in code. And that would be a copper bullet that benefits the entire computing profession. ■

Table 2. Grade average (converted to points) for database designs and models over time.

Statistic	Grade
Design average, first 21 case studies*	1.7
Design average, last 21 case studies	2.2
Model average, first 21 case studies	1.9
Model average, last 21 case studies	2.2

*The first 21 case studies are roughly from 1992 to 1997; the last 21 are roughly from 1997 to 2002.

Table 3. Relative average grades for vendor and in-house software.

Statistic	Grade
Design average, vendor	2.1
Design average, in-house	1.7
Model average, vendor	2.1
Model average, in-house	2.0

References

1. M. Blaha, *A Manager's Guide to Database Technology: Building and Purchasing Better Applications*, Prentice Hall, 2001.
2. M. Blaha, "On Reverse Engineering of Vendor Databases," *Proc. Working Conf. Reverse Eng.*, IEEE CS Press, 1998, pp. 183-190.
3. R. Box and M. Whitelaw, "Experiences When Migrating from Structured Analysis to Object-Oriented Modeling," *Proc. Australasian Computing Education Conf.*, ACM Press, 2000, pp. 12-18.
4. B.C. Behrens and R.R. Levary, "Practical Legal Aspects of Software Reverse Engineering," *Comm. ACM*, Feb. 1998, pp. 27-29.

Michael Blaha *is a partner of Modelsoft Consulting Corp., a consulting and training company based in Vancouver, BC, and is* Computer*'s area editor for databases and software. His research interests include object-oriented technology, modeling, system architecture, database design, enterprise integration, and reverse engineering. Blaha received a PhD in chemical engineering from Washington University and is a member of the IEEE Computer Society. Contact him at blaha@computer.org.*