It may seem hard to choose between two software products, but studying their databases can reveal a clear winner and save you years of grief.

Michael R. Blaha

The Case for Reverse Engineering

Most development methods stress forward engineering, in which you conceive, analyze, design, and implement an application. This is fine except that these days many enterprises, especially small ones, don't want all that trouble and cost. If your organization is like most, you want software you can plug in with a minimum of hassle. And you want the software to work smoothly with your existing systems. When you buy software, however, how do you know it's structurally sound?

Increasingly, organizations are looking for ROI guarantees. While there really aren't any hard and fast solutions, you can greatly minimize your headaches by reverse engineering potential products—literally picking them apart. It only takes a few weeks and is well worth the extra insight you'll get in return.

Inside

WHAT YOU DON'T SEE IS WHAT YOU GET

Perhaps the greatest argument for reverse engineering is that it keeps vendors accountable. Companies routinely assess products to determine their cost, functionality, user interface, and vendor stability. Reverse engineering lets you assess a product's intrinsic quality as well. It's amazing how many people just assume the software they buy is correct. Yet about half the databases we have studied at OMT Associates have major design errors. What's worse, a poor quality database design is a red flag to the poor quality of the overall software. For example, unnecessary redundancy can produce a range of nasty errors, such as the same person appearing on two forms with inconsistent phone numbers and addresses.

Reverse engineering also makes it easier to convert your data to the new system. Most organizations have to migrate large quantities of data with each new application. A side benefit is that you have a natural window for addressing nagging problems like duplicate personnel and payroll systems from that last business merger. You can't do any of this if you don't understand the logical correspondence between the old database and the new one.

GATHERING INFORMATION

The most productive way to reverse engineer a database is to build a model that conveys the software's scope and intent. For a clean database, you'll most likely want to prepare a model; for a flawed and poorly documented database, it may be best to stop after studying the style and quality of the database structure. At OMT Associates, we generally construct models—usually expressed as Unified Modeling Language (UML) object models.

As you gather information to build the model, consider as many sources of information as you can find, and remember that the number and kind of sources will vary widely across problems. The database structure is typically the dominant input because it specifies the data structure and many constraints precisely and explicitly. The structure varies according to the kind of DBMS, both by paradigm and product. Relational DBMSs

1520-9202/99/$10.00 © 1999 IEEE

Resources

Peter H. Aiken, *Data Reverse Engineering*, McGraw-Hill, New York, 1996—One of the few books available on database reverse engineering, deals mostly with project management issues.

Michael Blaha and William Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Upper Saddle River, N.J., 1998—Presents an object-oriented approach to conceiving, analyzing, designing, and implementing database applications. Addresses both forward and reverse engineering.

Kathi Hogshead Davis, "August-II: A Tool for Step-by-Step Data Model Reverse Engineering," *Proc. Second Working Conf. Reverse Engineering*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 146-154—Davis, a prominent research scientist in database reverse engineering, describes one of her tools.

Jean-Luc Hainaut et al., "Database Evolution: The DB-MAIN Approach," *Proc. 13th Entity-Relationship Conf.*, Springer-Verlag, New York, 1994, pp. 112-131—Hainaut is one of, if not *the*, leading academic researchers in database reverse engineering. This is one of many informative papers.

declare much table structure and can be readily inspected. Object-oriented DBMSs declare less, so it is more difficult to analyze their structure. Network and hierarchical DBMSs (such as Codasyl and IMS, respectively) also express fewer constraints.

Don't panic if information is missing; this is a common situation. Many databases have been optimized or were poorly designed to begin with. Most relational databases lack formal definitions of foreign keys (referential integrity). The designer may or may not define nullability for attributes, either because of sloppiness or just the desire to avoid the DBMS facilities for handling nulls. In these cases, just rely on conjecture and try to work around it in piecing together the database structure. For example, we often figure out foreign keys by finding one or more attributes that have the same name and data type as some primary key. Names and abbreviations are often mnemonic. In a system that tracks time spent on a project, "proj" probably means project and "ver" probably means version.

Data. You may have only a printed copy of the database structure and some explanation of what the database is supposed to do. In some cases, especially for large databases, you can sample the data to reach tentative conclusions and then explore the full database for verification. Examination cannot prove many propositions, but the more data you have, the more solid your conclusion will be. Remember, only one countercase disproves a hypothesis. It helps to keep a log of your observations as you study a database, noting any unusual aspects and pending questions. The log can be an invaluable tool in helping you and others recall design decisions and rationale later on.

A database structure may be badly flawed, yet the data may still be good. A thorough application program or disciplined users may give you better quality data than the database enforces. On the other hand, I've seen structures so badly flawed that the database could not possibly store correct data. Not surprisingly, these applications were failures.

Views and procedures. Normally you won't have programming code for a vendor product. But you may very well have access to views and stored procedures. For example, a join of two fields might indicate a relationship between the foreign key and the primary key. You may be able to trace a variable through a stored procedure and get some insight into the database structure.

Forms and reports. Suggestive titles and layouts can help you understand a database. Form and report definitions are especially helpful if you have their binding to database structures and queries. Alternatively, you can enter known synthetic data and trace the dataflow from the user interface through the database.

Documentation. Problems vary in the quality, quantity, and kind of documentation. User manuals are especially helpful. You can browse a user manual and quickly develop a model to guide reverse engineering. Sometimes the documentation for an application includes definitions of major constructs.

Application. If you understand an application well, you are, of course, in a much better position to make inferences. Take advantage of application experts who can answer specific questions. You may also be able to leverage models from related applications.

STEP BY STEP

We typically organize reverse engineering into three phases—implementation recovery, design recovery, and analysis recovery. This can take some adjusting because these phases are the inverse of what you see in most forward engineering efforts (analysis, design, and implementation).

In *implementation recovery*, you quickly learn about the application (typically a few hours of work), which helps you understand the evolving model. You then enter the database structures (either manually or automatically) into a model editor, representing each table as a class or entity type. We use the Rational Rose or Paradigm Plus editors. The columns of the tables are the attributes of the classes. Note nullability and data types, as well as primary keys, candidate keys, foreign keys, and any indexes. Do not include inferences at this point. It is better to have the initial model purely reflect implementation.

During *design recovery*, you undo the mechanics of the database structure. Again, you should stick with straightforward steps and defer most conjecture and interpretation to the next phase. Typically, you can study the database structure without help from application experts. The main goal is to resolve foreign-key references (such as references from one table to another). Matching names and data types can suggest references of a foreign key to a primary key. You may need to augment your study of database structure with experiments on the data. Unless the database has something odd, the observed values for a suspected foreign key should be a subset of those for the referenced primary key. Be sure to note any enumerations in the database or associated documentation.

Finally, during *analysis recovery* (also called data structure conceptualization), you interpret the model, refine it, and make it more abstract. This is where you remove the design artifacts and eliminate any database errors. Be sure to get rid of all redundant information or mark it as such. It's also a good time to look at the model. Is it readable and expressive enough? Reconcile the results of reverse engineering with models of other applications and documentation. Show your model to application experts and incorporate their advice.

Don't expect to proceed through these phases in a straight line. You'll move back and forth a lot. Also, not all parts of the database will be in the same phase at the same time because you'll be studying them separately.

A reverse-engineering expert should be able to complete all three phases in a few days or, at most, weeks. On average, it takes me about three weeks to evaluate a clean relational database with 500 tables. A novice will probably take three to four times longer, but each person should determine his own multiplier. Start with an arbitrary number, say four times longer, and prepare for an overrun. Monitor the time it takes to do various tasks and adjust the original estimate as needed.

The time you take also depends on the database. Size—the number of tables and attributes in a relational database, for example—is an obvious factor, but the database paradigm also affects the effort. Databases with explicitly declared constraints are typically less difficult to reverse engineer, so relational databases are usually more straightforward than Cobol files or IMS, Codasyl, and object-oriented databases. And, of course, a clean database—one with a coherent design, consistent naming, and resolvable foreign keys—is considerably easier to study than a database riddled with errors.

When you reverse engineer a database, you have to know when to stop. A simple rule is, stop when you are no longer productive. Sometimes, it helps to set a time before

Grading a Database

As you evaluate a database, it helps to use some rating system to measure design consistency and quantify the design and modeling errors. Grades are nice because they already have a set of connotations; it's not hard for management to understand that F means the database is pretty bad.

Here is a grading system we use at OMT Associates.

	Explanation	Types of flaws
A	Clean	No major flaws. Style is reasonable and uniformly applied.
B	Structural flaws, but they don't affect the application	Flaws can be repaired without much disruption. Data types and lengths not uniformly assigned. Null constraints not used to distinguish optional and required fields. Candidate keys and enumerations not defined. Columns have cryptic names like Cell 123
C	Major flaws that affect the application (bugs, low performance, can't be extended)	Flaws are not easy to fix. Inconsistent approach to identity. Indexing haphazard. Many foreign keys lack indexes, and some indexes are subsumed by others. Foreign keys have mismatched data types.
D	Severe flaws that compromise the application	Primary keys not readily apparent. Much unnecessary redundant data. Stores extensive binary data (compiled programming language data structures), which subverts data declaration.
F	Appalling (the application won't run properly or runs only because of brute-force programming)	Deep conceptual errors in underlying model, gross design errors, or both. The creativity behind these flaws stretches the imagination.

What Is the Unified Modeling Language?

The Unified Modeling Language (UML) is a general-purpose visual modeling language created by Grady Booch, James Rumbaugh, and Ivar Jacobson of Rational Software Corp. The programming community has already endorsed the UML, and the language is making inroads with the database community, as well.

For example, the Object Management Group (OMG) has made UML a standard notation for modeling systems, in part because it supports a variety of models, including models of structure, function, behavior, and architecture. The UML model of most interest in reverse engineering a database is the model of structure, or object model. The object model is essentially just another dialect in the many variations of the entity-relationship model—the dominant approach to modeling databases. The object model makes it easier to do reverse engineering because it is adept at representing both abstract conceptual models and models with implementation decisions.

UML models are helpful because they let you develop a thorough understanding of real-world requirements. They are also concise and readily map to implementation constructs, such as relational-database tables. Finally, the models also let you represent your application without concern for the details of implementation.

A good place to begin reading more about UML is the *UML User's Guide* (G. Booch et al., Addison Wesley Longman, 1999), which gives some foundational details. Those who want to study more can continue with the more intense *UML Reference Manual* (J. Rumbaugh et al., Addison Wesley Longman, 1999), which explains much of the theory and deep underlying concepts.

you begin and stop when you reach that limit, even if that means you get only some results. Pace yourself. If you can justify only a small amount of time, consider shortcuts like looking at only part of the database.

TOOLS AND SKILLS

Uncertainty is endemic to reverse engineering; it is not always obvious how a tool should manage uncertainty and convey it to the user. Furthermore, a tool must be able to combine automation with user inferences, and it is not obvious how to devise a metaphor for user assertions that is both powerful and easy to understand.

Some commercial products nominally provide reverse engineering, but typically support only the well-designed databases that are least in need. I advise looking to the current research tools. DB-MAIN, developed by Jean-Luc Hainaut and his colleagues at the University of Namur, Belgium, is the best reverse-engineering tool for databases I've seen to date. Kathi Davis of Northern Illinois University has also devised a substantial tool. You need not confine yourself to one tool; at OMT Associates, we have combined model editors, automated model input, SQL queries, and AWK scripts in various ways.

Whatever the tool, it won't replace skilled users. Reverse engineering a database requires a formidable mix of skills—one reason this approach is not more popular. Unfortunately, this sets up a kind of catch-22. Because reverse engineering is only sporadically applied, it's hard to keep current, let alone proficient, with new techniques. Large organizations can cope by training select individuals in depth rather than trying to give a little something to everyone. Most small organizations should outsource reverse-engineering tasks.

Modeling, especially object or entity-relationship modeling, is a prerequisite to reverse engineering. This hangs up a lot of people, who find it hard to abstract and realize an application indirectly through models rather than directly through coding. Another core skill is database design. As reverse engineers, you must be able to recognize design idioms quickly and translate them into modeling constructs—especially for large databases. This implies that you must first thoroughly understand forward-engineering rules. Proficiency with SQL also helps if you are exploring data in relational databases. The ability to interpret database designs is even more critical in light of the many ad hoc optimizations and flaws you are likely to encounter.

Inherent in these core skills are two soft skills: focusing and self-confidence. You must be satisfied with quickly obtaining 80 percent to 90 percent of the available information, and not get mired in details or the slavish desire to go the extra 10 to 20 percent—it's seldom worth the effort. You must also be confident in your ability to second-guess other designers. You'll see some odd, incorrect, and baffling designs. Uncertainty is endemic to reverse engineering, but you'll have to deal with it. You will gain confidence as you practice reverse engineering and encounter different situations.

To promote these skills in your organization, focus your initial training on the mainstream skills—modeling and database design—and then go on to specific training in the reverse-engineering process. Currently, there are few sources for reverse-engineering training, so you may have to grow the skills on your own. Select a few self-starters, send them to selected conferences (the IEEE Working Conference on Reverse Engineering—http://www.cc.gatech.edu/

reverse/wcrecfp.html—is a good choice), and assign them a few databases to work on. Be prepared for some missteps and rework along the way. If possible, bring in an outside mentor.

A TALE OF TWO DATABASES

To give you some idea of what you can learn from reverse engineering, I offer two anecdotes from the OMT Associates files. I mention no specific vendors or packages both out of courtesy to the vendors and in the hope that current versions of the software may not contain the same flaws.

These stories are fairly typical of what we find—major design errors in about half the databases and conceptual errors in a fourth. Design errors involve the mechanics of a database. Given a sound model, it should be straightforward to prepare a database design. However, designers are human, and if databases are outside their expertise, they may not realize the potential for error. Conceptual errors are usually due to insufficient modeling during development. These errors are especially nasty because typically you must deeply understand the application to detect them and repair them.

Comparing product finalists

Our client asked us to evaluate two product finalists that were frameworks for building mechanical engineering applications. The favored product had been marketed by a big name for many years. The vendor of the alternative product was not well known, and the product had been out for only a few years.

After prototyping a small test application with each finalist, we found that the favored product required more training and was more difficult to understand. It also required about triple the development effort and was slower. Nevertheless, we were still uncertain because the negative findings could be the result of assigning less experienced support staff to our project.

When we reverse engineered the two databases, however, we got a much clearer picture. The favored product had a solid database design, but badly flawed conceptualization. The alternative product had both a clean model and clean database design. It became obvious that our difficulty with training and implementation was caused by flaws in the favored product. Clearly, the alternative product was a better choice.

It took us about one person-week to reverse engineer the favored product of about 150 tables. It took us about two person-weeks to reverse engineer the alternative product of about 50 tables. For the favored product, we just studied the database structure because we were unable to construct a complete model. In the alternative product, the high level of abstraction inherent in a framework made the resulting model more difficult to understand and construct than an ordinary application. We also analyzed the data in detail, formulating hypotheses about the model and testing them against the actual database.

In this case, reverse engineering let us directly compare the intrinsic quality of the two products and kept us from choosing an inferior product. We were truly undecided before the reverse engineering. The development and deployment budget for the client application was about $500,000. I estimate that the choice of the better product saved at least $50,000. We may have still chosen the better product without reverse engineering, but it would have required a minimum of a few months of additional prototyping. The reverse engineering was well worth the three person-weeks of effort. This large payout is typical when you are selecting a vendor product for a major application.

Mitigating software flaws

In this project, we reverse engineered a database for a customer and assigned it an "F," because the database design was a mess. The vendor also had an unorthodox software architecture. Nevertheless, our client chose to purchase this product, a decision I endorsed despite the awfulness of the design because there were few suitable products out there, and even this flawed application could deliver business benefit. Furthermore, the product had some technical merit apart from its flaws. (We did recommend that our client shift to a better product if one becomes available.)

However, the reverse-engineering results made our client aware of the database's negative qualities, and we prepared a strategy to mitigate the consequences. Essentially, the database structure did not assure data quality, which made it more difficult to load legacy data and gave the database a tendency to accumulate bad data. The database was also difficult to extend.

It took us about two person-weeks to reverse engineer this database of about 100 tables. We studied both the data-

Checklist for Outsourcing

If you decide to outsource reverse engineering, make sure your reverse-engineering contractor is

- ☑ Proficient with both modeling and databases, especially relational databases.
- ☑ Able to show you outputs from some past projects and to quantify the benefits the outputs provided.
- ☑ Willing to commit to a fixed-price contract. Reverse engineering does not have to be perfect or complete, and you don't want the scope to get out of hand.
- ☑ Conversant with key published work. Reverse engineering is an advanced skill; it is questionable that someone can be capable without knowing what's in the literature.

Comparing Forward and Reverse Engineering

As the table below shows, reverse engineering requires a completely different mind-set from forward engineering. To adapt to this change, it helps to understand key points of contrast.

Forward engineering	Reverse engineering
Given requirements, develop an application	Given an application, deduce tentative requirements
More certain; developer has requirements and must deliver an application that implements them	Less certain; an implementation can yield different requirements, depending on the reverse engineer's interpretation
Prescriptive; developers are told how to work	Adaptive; reverse engineer must find out what the developer actually did
More mature; skilled staff readily available	Less mature; skilled staff sparse
Time-consuming (months to years of work)	Can be performed 10 to 100 times faster than forward engineering (days to weeks of work)
Model must be correct and complete or the application will fail	Model can be imperfect; salvaging part of the information is still useful
Process of creation, the system must change	Process of examination, the system need not change

Goals. Don't be discouraged by the imperfection of reverse engineering. It is worth investing a modest amount of time to extract 80 percent of the meaning of an existing database. Reverse engineering injects reality into modeling and is more efficient than starting from scratch. You can use the typical forward-engineering techniques (such as interviewing knowledgeable users) to obtain the remaining 20 percent. This 80 percent goal contrasts sharply with the aim for perfection in forward engineering, and it takes some getting used to.

Process. In contrast to forward engineering, in which developers are explicitly told requirements, reverse engineering yields hypotheses you may have to reconsider as you learn more about a problem. Remember that the more you learn about an application, the firmer your hypotheses become. The uncertainty factor also means greater flexibility. You'll need to keep adjusting the process to fit the problem because problem styles and the amount of information you can get will vary widely. Use your wits, and approach the task as if you were solving a large puzzle. You're going to see some pretty odd stuff. Database designers, even the experts, occasionally violate database design and can use unexpected constructs. You may be able to look at part of the database structure to deduce the design strategy. Database designers are typically consistent, regardless of what they choose to do, even if that is to consistently violate good design practice.

Solutions. There's no single answer because there can be more than one plausible interpretation. In general, the mapping between object models and database structures is many-to-many because when you reverse engineer a database, each interpretation of the structure and data can yield a different object model. The more information you have, however, the more quickly the interpretations among reverse engineers will converge. In forward engineering, you start with a single model, but different implementation decisions can yield different database structures. With reverse engineering the differences among the models can be profound, depending on the uncertainty and the degree of interpretation required. With forward engineering, there can be multiple database structures, but they mean the same thing; the differences largely reflect developer preferences.

base structure and documentation in preparing our model. The unorthodox architecture complicated our task and impeded our efficiency.

Our client was able to use the results of reverse engineering to negotiate for price, terms, and conditions with the vendor. It also helped our client understand the vendor's strange architecture, so they could communicate to the vendor about the product.

THE CARROT AND THE STICK

As these stories show, reverse engineering is more than just a process for studying databases; it has profound implications for software development in general. You might view the reverse engineering of vendor databases as a creative response to the so-called software crisis. The current approach of preaching methodology and discipline isn't working. Software quality and development productivity still badly lag behind our desires. Database reverse engineering gives us both a carrot and a stick. The flaws and excellence of various products become more obvious and more heavily influence product success and failure. For a large corporation, the cost of the evaluation (as little as a few person-weeks) is trivial compared to the millions spent buying and deploying new software. ■

Michael R. Blaha is a principal of OMT Associates Inc., a training and consulting firm specializing in modeling, database design, and reverse engineering.

Contact him at blaha@computer.org or at OMT's Web site http://www.omtassociates.com.

How to Reach IT Pro

Writers

We welcome submissions. For detailed information, write for a Contributors' Guide (itpro@computer.org) or visit our Web site: http://computer.org/itpro.

Letters to the Editor

Please provide an e-mail address or daytime phone number with your letter. Send letters to *IT Pro* Letters, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; fax (714) 821-4010; itpro@computer.org.

Missing or Damaged Copies

If you are missing an issue or received a damaged copy, contact membership@computer.org.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.
To buy a reprint, send a query to itpro@computer.org or a fax to (714) 821-4010.