Presented at 6th Working Conference on Reverse Engineering
Atlanta, Georgia — October 1999

An Industrial Example of Database Reverse Engineering

Michael Blaha

OMT Associates Inc., Chesterfield, Missouri 63017 USA (blaha@computer.org)
www.omtassociates.com

Abstract

This paper presents an industrial example of database reverse engineering. The example has been abridged so that it fits within a paper. Also some of the field names have been disguised as a courtesy to the source company. Nevertheless, the example is real and illustrates the kinds of mistakes and poor design that are often found in practice.

1. Introduction

At the past few WCRE conferences there have been several questions from program reverse engineering experts who are trying to understand what is involved with database reverse engineering. This paper presents industrial database code, analyzes it, and shows the kinds of problems and errors that occur. At first glance some readers might be appalled by the sloppiness of the design. But this is not the worst practice I have found. On a grading scale of "A" (best) to "F" (worst), I would give it a grade of "D". [3]

Another motivation for the paper is to add an example to the literature. Few database reverse engineering examples have been published. Examples are important because they document current practice, demonstrate reverse engineering techniques, and provide grist for tool developers.

The example is abbreviated. The complete database code is about five times longer than that shown here. However, the excerpt is representative of the full code, preserving the content and the style. The purpose of the application is to import mainframe data and make it available for use on other platforms. The mainframe data is stored in COBOL files and the ported data is stored in a relational database.

The next section of the paper presents the source database code and notes peculiarities and flaws. Section 3 then briefly summarizes a database reverse engineering process. The remainder of the paper applies the process and presents some overall conclusions.

2. Source Code and Commentary

Tables 1 through 4 present the source code for the example. The statements are executed to create empty relational database tables.

2.1 *Location* Table

Table 1 presents code for the *location* table. The fields of the table are listed after the first left parenthesis. The table has 15 fields in all: *location_num* through *location_cas_gross_profit_dol*. Each field has a specified data type—integer (*number* data type with parameter of total digits), fixed point (*number* data type with parameters of total and fractional digits), or string (*varchar2* data type with a length). Constraints on the table are shown at the end of the table definition code. The *primary key* constraint is explained later in this section.

```
CREATE TABLE location
  (location_num           NUMBER(3)
  ,location_name          VARCHAR2(15)
  ,location_address_1     VARCHAR2(30)
  ,location_address_2     VARCHAR2(30)
  ,location_address_3     VARCHAR2(30)
  ,location_address_4     VARCHAR2(30)
  ,location_address_5     VARCHAR2(30)
  ,location_group_code    NUMBER(2)
  ,location_business_type VARCHAR2(1)
  ,location_tot_bus_sales_dol
     NUMBER(11,2)
  ,location_gross_profit_dol
     NUMBER(11,2)
  ,location_atv_sales_dol NUMBER(11,2)
  ,location_atv_gross_profit_dol
     NUMBER(11,2)
  ,location_cas_sales_dol NUMBER(11,2)
  ,location_cas_gross_profit_dol
     NUMBER(11,2)
  ,CONSTRAINT ic_location_00 PRIMARY KEY
     (location_num ) ) ;
```

Table 1 Code to create *location* table

Note that the field names have a prefix of the table name. This is a common style; some applications use an abbreviation instead of a lengthy table name. Many tools, such as the DB-MAIN tool [5], support prefixes. During forward engineering, the developer can add a prefix to the field names of a model. During reverse engineering, the developer can remove prefixes to reduce clutter. For brevity, the discussion in the remainder of the paper omits most field prefixes

By default, any of the fields in a relational database table may have a null value. (***Null*** is a special value denoting that a field is unknown or not applicable for a given record.) The default is appropriate for most fields, since they hold optional data.

However, proper database design practice is to define *not null* constraints for all fields that participate in primary and candidate keys. (A ***candidate key*** is one or more fields that uniquely identify the records in a table. The set of fields in a candidate key must be minimal; no field can be discarded from the candidate key without destroying uniqueness. A ***primary key*** is an arbitrarily chosen candidate key used to reference records preferentially.) In this regard, the code is remiss. The *location* table has a primary key clause near the end of the code. However, field *location_num* lacks a *not null* constraint.

Also note the multiple fields for address data. Strictly speaking, this satisfies database design theory, but is sloppy nonetheless. Consider the fragments of data in Figure 1. 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

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 Poor design practice: Anonymous fields

The *location* table has several fields with parameters embedded in their name: *tot_bus_sales_dol*, *gross_profit_dol*, *atv_sales_dol*, *atv_gross_profit_dol*, *cas_sales_dol*, and *cas_gross_profit_dol*. (The full *location* table has about 15 parameterized fields.) One parameter denotes the kind of business (*atv* and *cas*) and the other denotes the kinds of dollars (*gross profit* and *sales*). Even though it conforms to theory, this is a questionable design practice. It is verbose to list the various combinations of parameters. A better approach is to restructure the data and use multiple records (rather than multiple fields) to express the different combinations. (See Section 6.)

2.2 *Customer* Table

Table 2 presents the *customer* table. This table also has prefixes for field names with the exception of the first field in the table. Such irregularities are not a good idea. Two constraints are shown at the end of the table—one is a primary key and the other is a candidate key (denoted by the *unique* keyword).

The address information is poorly structured. There are two groups of address fields: residence address and billing address. Address data depends on both the primary key

```
CREATE TABLE customer
  (id_customer              NUMBER(9)
  ,customer_account_num     NUMBER(7)
  ,customer_location_num    NUMBER(3)
  ,customer_prim_salesperson_num
     NUMBER(4)
  ,customer_name            VARCHAR2(40)
  ,customer_address1        VARCHAR2(30)
  ,customer_address2        VARCHAR2(30)
  ,customer_city            VARCHAR2(30)
  ,customer_state           VARCHAR2(2)
  ,customer_country         VARCHAR2(25)
  ,customer_zip_code1       NUMBER(5)
  ,customer_zip_code2       NUMBER(4)
  ,customer_type            VARCHAR2(1)
  ,customer_bill_address1   VARCHAR2(30)
  ,customer_bill_address2   VARCHAR2(30)
  ,customer_bill_city       VARCHAR2(30)
  ,customer_bill_state      VARCHAR2(2)
  ,customer_bill_country    VARCHAR2(25)
  ,customer_bill_zip_code1 NUMBER(5)
  ,customer_bill_zip_code2 NUMBER(4)
  ,customer_contact_name    VARCHAR2(30)
  ,customer_contact_position
     VARCHAR2(30)
  ,customer_contact_phone   VARCHAR2(20)
  ,customer_dun_num         NUMBER(9)
  ,customer_last_activity_date DATE
  ,customer_year_started    NUMBER(4)
  ,customer_tax_status      VARCHAR2(1)
  ,customer_status          NUMBER(1)
  ,CONSTRAINT ic_customer_00 PRIMARY KEY
     (id_customer )
  ,CONSTRAINT ic_customer_01 UNIQUE
     (customer_account_num,
     customer_location_num ) ) ;
```

Table 2 Code to create *customer* table

(the identification of the customer) and the kind of address (residence or billing). It would be much better to refer to address data in a separate table.

Similarly, the contact data is unsound. 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 the database design rule called 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 could enforce uniformity of position and phone. Section 6 makes such a redesign to the tables.

There is some inconsistency with data types. In the *location* table, location_name is VARCHAR2(15). In the *customer* table customer_name is VARCHAR2(40) and *customer_contact_name* is VARCHAR2(30). All these fields are names and a cleaner design would use the same length.

Also note the inconsistency between the *location* and *customer* tables in handling address data. The *location* table stores address data with anonymous fields (*address_1* through *address_5*) while the *customer* table has specific fields (*address1*, *address2*, *city*, *state*, *country*, *zip_code1*, and *zip_code2*). A better design would handle address data in a uniform manner. As Section 2.1 mentioned, the use of specific fields is the better approach.

The *country* field is intended to distinguish between the U.S. and Canada. But the developer made a mistake with zip codes. U.S. zip codes are all numbers; Canadian zip codes are numbers and letters.

2.3 *Contract* Table

Table 3 presents the *contract* table. The name of this table is a misnomer. The table actually combines contract data with contract revision data. (A contract has many contract revisions.) I suspected the combination based on the field names and confirmed the suspicion with the designers. The *contract* table violates second normal form which warns designers not to combine distinct kinds of data.

Note the dual source of identity (*id_contract*, *contract_num*). This is an especially gross error. Field *id_contract* is an artificial sequence number. (It is a common and reasonable design practice to define artificial sequence numbers for use as primary keys.) Field *contract_num* is the number of the contract, a field with business meaning. The primary key is *id_contract* + *revision_num* and a candidate key is *contract_num* + *revision_num*. Field *id_contract* is in a one-to-one relationship with *contract_num*.

I talked to the developers and discovered that the dual identity was caused by an attempt to simplify data loading. The developers intended to store data in the various tables

```
CREATE TABLE contract
  (id_contract              NUMBER(9)
  ,contract_revision_num    NUMBER(4)
  ,contract_num             NUMBER(8)
  ,contract_prev_num        NUMBER(8)
  ,contract_status_code     NUMBER(1)
  ,contract_type            VARCHAR2(2)
  ,contract_ident_num       NUMBER(4)
  ,contract_sales_type      NUMBER(1)
  ,contract_location_num    NUMBER(3)
  ,id_customer              NUMBER(9)
  ,contract_current_revision_ind
     VARCHAR2(1)
  ,contract_revision_date   DATE
  ,contract_start_date      DATE
  ,contract_duration        NUMBER(3)
  ,contract_project_num     NUMBER(5)
  ,contract_project_name    VARCHAR2(25)
  ,contract_project_location
     VARCHAR2(25)
  ,contract_salespsn_admin_num NUMBER(4)
  ,contract_tot_amount      NUMBER(11,2)
  ,contract_xpctd_grss_prof_dol
     NUMBER(11,2)
  ,contract_act_gross_profit_dol
     NUMBER(11,2)
  ,contract_est_gross_margin_pct
     NUMBER(4,1)
  ,contract_act_markup_pct NUMBER(4,1)
  ,contract_est_costs_dol  NUMBER(11,2)
  ,contract_budgeted_costs_dol
     NUMBER(11,2)
  ,contract_act_costs_dol  NUMBER(11,2)
  ,CONSTRAINT ic_contract_00 PRIMARY KEY
     (id_contract,contract_revision_num)
  ,CONSTRAINT ic_contract_01 UNIQUE
   contract_num,contract_revision_num));

CREATE INDEX ic_contract_02 ON
  contract ( id_customer ) ;
```

Table 3 Code to create *contract* table

first and then assign values to the artificial *id* fields. Rather they should have assigned *ids* as data was stored and used *ids* as the only source of identity. In essence, they should have designed the database correctly and complicated the code for loading data.

The developers defined an index on the *id_customer* field of *contract*. Indexes are special data structures that augment a table and speed the response to certain queries. Database designers often index foreign key fields so that joins of common data are fast and efficient. Consequently, indexes are a reverse engineering clue that might indicate a foreign key. Once again the database structure has the flaw

of inconsistency because other foreign keys are not indexed (*prev_num* in the *contract* table and *location_num* in the *customer* table). Good design practice is to index all foreign keys.

Field *prev_num* refers to the previous *contract_num* and is a foreign key. I inferred the foreign key based on the suggestive field names and the matching data types. The developers confirmed my inference. The foreign key refers to only part of a candidate key which is another major error. This is a consequence of mixing contract and contract revision data. For a correct database design, each foreign key should refer to an entire candidate key (and better yet to a candidate key which has been designated as the primary key).

The *contract* table also has a violation of third normal form. The project name and project location depend on the project number which in turn depends on the primary key. A separate project table would rectify this problem.

The *contract* table has one last flaw. There is no good reason to store *location_num* in the *contract* table; it can be readily retrieved by joining the *contract* table to the *customer* table on *id_customer*. Theoretically, it is acceptable to store derived data. Practically, derived data should only be used when there is a compelling need to speed performance. The drawback of unnecessary redundancy is that it introduces the possibility of inconsistent data. Developers often make the mistake of including derived data for frivolous reasons.

2.4 *Contcost* Table

Table 4 presents the *contcost* table. The redundant references to *id_contract* and *contract_num* are consistent with the dual identity in the *contract* table.

Comparison of the primary key and unique constraints reveals that the primary key has more than a minimum number of fields. This is another major error. This flaw was also caused by the developers good, but misguided, intentions of facilitating data conversion. Developers should never distort a database to facilitate data conversion. A poor database design complicates programming and long term maintenance.

By inspection (comparing data types and constraints) *extra_num* in the *contcost* table seems to be the same as *revision_num* in the *contract* table. I confirmed this hypothesis with the developers.

Note the parallel data for estimating (*est*), re-estimating (*reest*), and actual (*act*) dollars and hours. There is an initial estimate of a contract, a re-estimate, and then the actual numbers. The two estimates are an artifact of the existing business process and subject to change. For example, in the future, there could be a single estimate or three estimates. A business loses flexibility by encoding such arbitrary business practices into the database structure. This leads to ossified systems and business impatience with the slow response of computing professionals to make changes. A better approach is to restructure the data with the different dimensions as fields of a table. (See Section 6.)

```
CREATE TABLE contcost
  (id_contract               NUMBER(9)
  ,contcost_extra_num        NUMBER(4)
  ,contcost_contract_num     NUMBER(8)
  ,contcost_contract_type    VARCHAR2(2)
  ,contcost_est_matl_dol     NUMBER(11,2)
  ,contcost_reest_matl_dol   NUMBER(11,2)
  ,contcost_act_matl_dol     NUMBER(11,2)
  ,contcost_est_a_lab_dol    NUMBER(11,2)
  ,contcost_reest_a_labexp_dol
     NUMBER(11,2)
  ,contcost_act_a_lab_dol    NUMBER(11,2)
  ,contcost_est_a_lab_hours NUMBER(7)
  ,contcost_reest_a_lab_hours
     NUMBER(7)
  ,contcost_act_a_lab_hours NUMBER(7)
  ,contcost_est_b_labexp_dol
     NUMBER(11,2)
  ,contcost_reest_b_labexp_dol
     NUMBER(11,2)
  ,contcost_act_b_labexp_dol
     NUMBER(11,2)
  ,contcost_est_b_lab_hours NUMBER(7)
  ,contcost_reest_b_lab_hours NUMBER(7)
  ,contcost_act_b_lab_hours NUMBER(7)
  ,contcost_other_b_labexp_dol
     NUMBER(11,2)
  ,contcost_reest_oth_labexp_dol
     NUMBER(11,2)
  ,contcost_act_other_labexp_dol
     NUMBER(11,2)
  ,contcost_est_other_lab_hours
     NUMBER(7)
  ,contcost_reest_other_lab_hours
     NUMBER(7)
  ,contcost_act_other_lab_hours
     NUMBER(7)
  ,contcost_est_proficiency NUMBER(11,2)
  ,contcost_reest_proficiency
     NUMBER(11,2)
  ,contcost_est_risk         NUMBER(11,2)
  ,contcost_reest_risk       NUMBER(11,2)
  ,CONSTRAINT ic_contcost_00 PRIMARY KEY
     (id_contract,contcost_contract_num,
     contcost_extra_num )
  ,CONSTRAINT ic_contcost_01 UNIQUE
     (contcost_contract_num,
     contcost_extra_num ) ) ;
```

Table 4 Code to create *contcost* table

3. Summary of Reverse Engineering Process

At this point we have learned quite a bit from the source code. Now it is appropriate to take the code and recast it as a model. A model provides a useful substrate for removing the errors and artifacts of implementation as well as considering enhancements.

The models in the remainder of the paper use the UML (Unified Modeling Language) notation. The UML is a clean, concise notation and a standard. It is as good a choice as any for database modeling.

It is helpful to organize reverse engineering into three phases that are the inverse of the typical forward engineering phases.

- **Implementation recovery**. First quickly learn about the application. Then enter the database structures into a model editor. You should tentatively represent each table as a class or entity type. During this phase you should defer any inferences. For reference purposes it is helpful to have an initial model that purely reflects the implementation.
- **Design recovery**. Undo the mechanics of the database structure. Typically, you can study the database structure autonomously and need not interact with application experts. The major purpose of design recovery is to resolve foreign key references (references from one table to another).
- **Analysis recovery**. Interpret the model, refine it, and make it more abstract. During analysis you remove the artifacts of design and eliminate any errors in the model. You should review your findings with application experts. Reconsider the model to improve its readability and expressiveness.

The primary input to database reverse engineering for this example was the database structure; this is the normal situation. I also benefitted from discussions with the application developers. There were no other inputs. The developers could not provide sample data because they had not yet populated the database. I did not ask for their programming code because it was incomplete and hobbled by their poor database structure.

The intended output of reverse engineering was an assessment of the current effort and a redesign of the database to fix its flaws. The redesign was never completed as Section 7 explains.

4. Model from Implementation Recovery

I typed the code from Tables 1 through 4 into a modeling tool. For brevity I do not show the model here. Section 2 has already presented some context about the application gleaned from studying the code and talking to developers.

5. Model from Design Recovery

Figure 2 shows the model after design recovery. There are four classes in the diagram: *Contract*, *Contcost*, *Location*, and *Customer*. A box is the UML notation for a class. (The box with *account_num* is another construct to be discussed.) A ***class*** describes objects with common fields, behavior, and intent. An ***object*** is a concept, abstraction, or thing that has meaning for an application. For example, Standard Widget and the Simplex company may be customers. Fields may be suppressed or displayed in the second portion of the class box.

Classes are related by ***associations***. The UML notation for an association is a solid line that may consist of a number of line segments. The diagram in Figure 2 has three associations: *Contract* to *Contcost*, *Contract* to *Customer*, and *Location* to *Customer*. The annotations at the ends of the associations denote ***multiplicity***—the number of objects of one class that may relate to a single object of an associated class. A *Contcost* object pertains to a single *Contract* (multiplicity "1"). A *Contract* may have at most one associated *Contcost* object (multiplicity "0..1"). A *Customer* may have many *Contracts* (multiplicity "*").

I carried all fields forward from the original code, except for foreign key fields which are converted to associations. For brevity Figure 2 does not display the fields of the *Location* and *Contract* classes. The *Location* class has all the fields shown in Table 1. The *Contract* class has all the fields shown in Table 3, except for *id_customer*; *id_customer* is a foreign key that has been converted into an association from *Contract* to *Customer*. I deferred resolution of *contract_prev_num* until analysis recovery.

The *Contcost* class has all fields shown in Table 4 except for the first three fields which clumsily refer to *Contract*. A *Contcost* has one contract revision. This is apparent by comparing the constraints on the *contract* and *contcost* tables and recalling that *contcost_extra_num* is the same as *contract_revision_num*. (Also recall that *Contract* combines contract and contract revision data. These two aspects are separated during analysis recovery.)

The *Customer* class has all fields shown in Table 2 except for *account_num* and *location_num*. These fields have been resolved to the qualified association: A location plus the qualifier *account_num* yields a single customer. The ***qualifier*** is a special kind of field that increases the precision of the association. The qualified association expresses the uniqueness constraint in Table 2. A location has many customers; the qualifier lets us tell the customers apart.

6. Model from Analysis Recovery

Figure 3 shows the partial results after analysis recovery. It is not obvious how to split the fields between contract and contract revision so I left the fields in the *Contract*

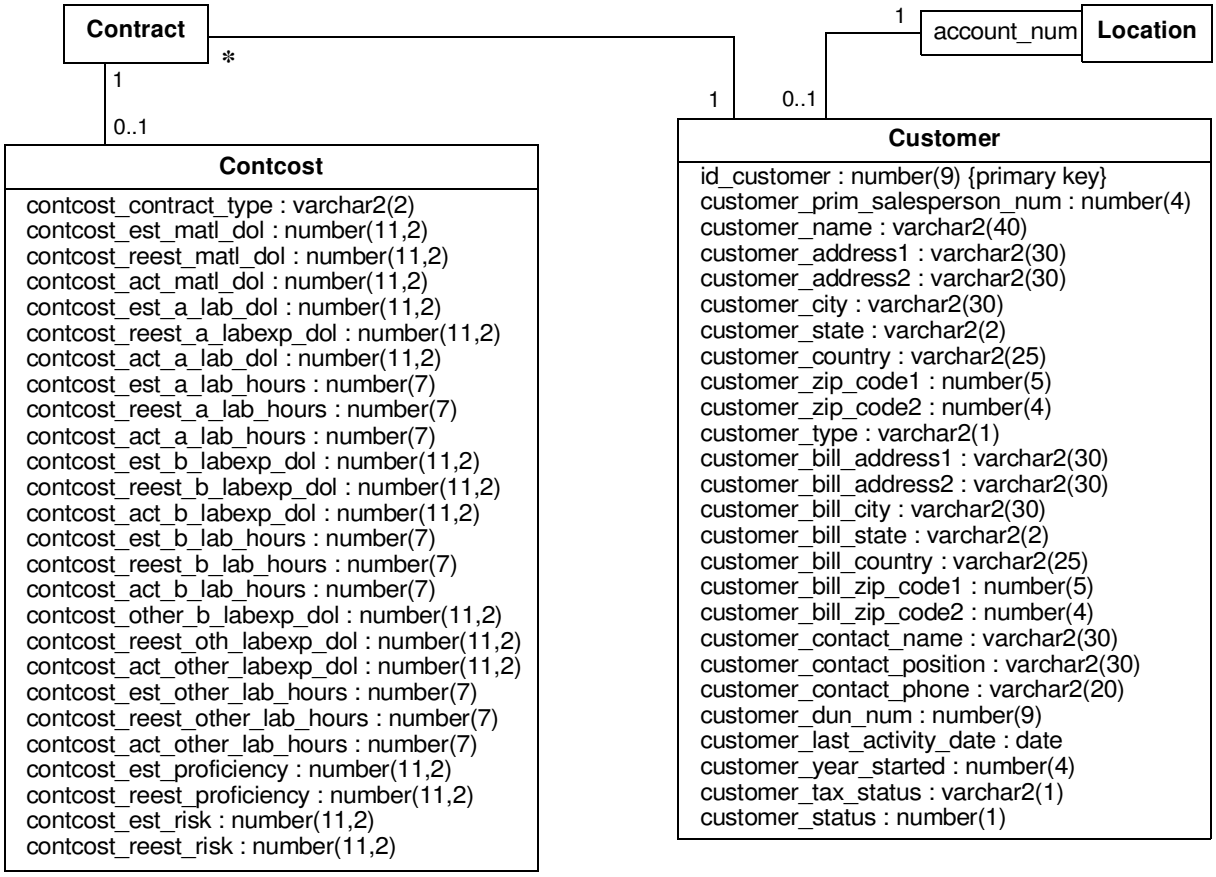

Figure 2 Model after Design Recovery

class. (The original developers confirmed the distinction between contract and contract revision, but they did not explain how to split the fields.)

I will briefly explain the model. A customer may have a billing address and a residence address. (Billing and residence are roles; a ***role*** is a usage of a class.) An address may be the residence of many customers and used to bill many customers. A customer may designate a person to serve as a contact. A person may be a contact for multiple customers.

A business location may serve many customers who are distinguished by an account number. Each business location has an address and it is possible (though unlikely) for an address to be used for more than one business location.

A location may have many financial values. Each value can be found with the combination of a location, financial category, and financial unit. The *LocationFinancialValue* class resolves the parameterized fields in the *location* table in a more elegant manner. Instead of storing multiple values in a *location* record, the revised model stores multiple *LocationFinancialValue* records. The records for *FinancialCategory* include *total_business*, *atv*, and *cas*. The records for *FinancialUnit* include *sales* and *gross_profit*. With the revised structure, it is easy to add a financial category or a financial unit.

A contract may have many contract revisions, which are differentiated with a revision number. Each contract revision pertains to a specific contract. *CostValue* resolves the parameterized fields for *ContractRevision* in a similar manner to that for financial value. Each cost value is determined by the combination of a contract revision, a cost category, a cost quality, and a cost unit.

CostQuality has the values *actual*, *estimate*, and *reestimate*; additional values can be easily added. *CostCategory* has the values *risk*, *proficiency*, *other lab*, *other lab expense*, *a lab*, *b lab*, and *material*. (I am not sure what all these cost categories mean; this would have to be clarified before completing the analysis model.) *CostUnits* has the values *dollars* and *hours*.

A contract may have a previous contract indicated. This is a questionable representation. The existing model essentially has a chain of previous dependencies which could be tedious to traverse. A better model would instead relate a collection of contracts to each other.

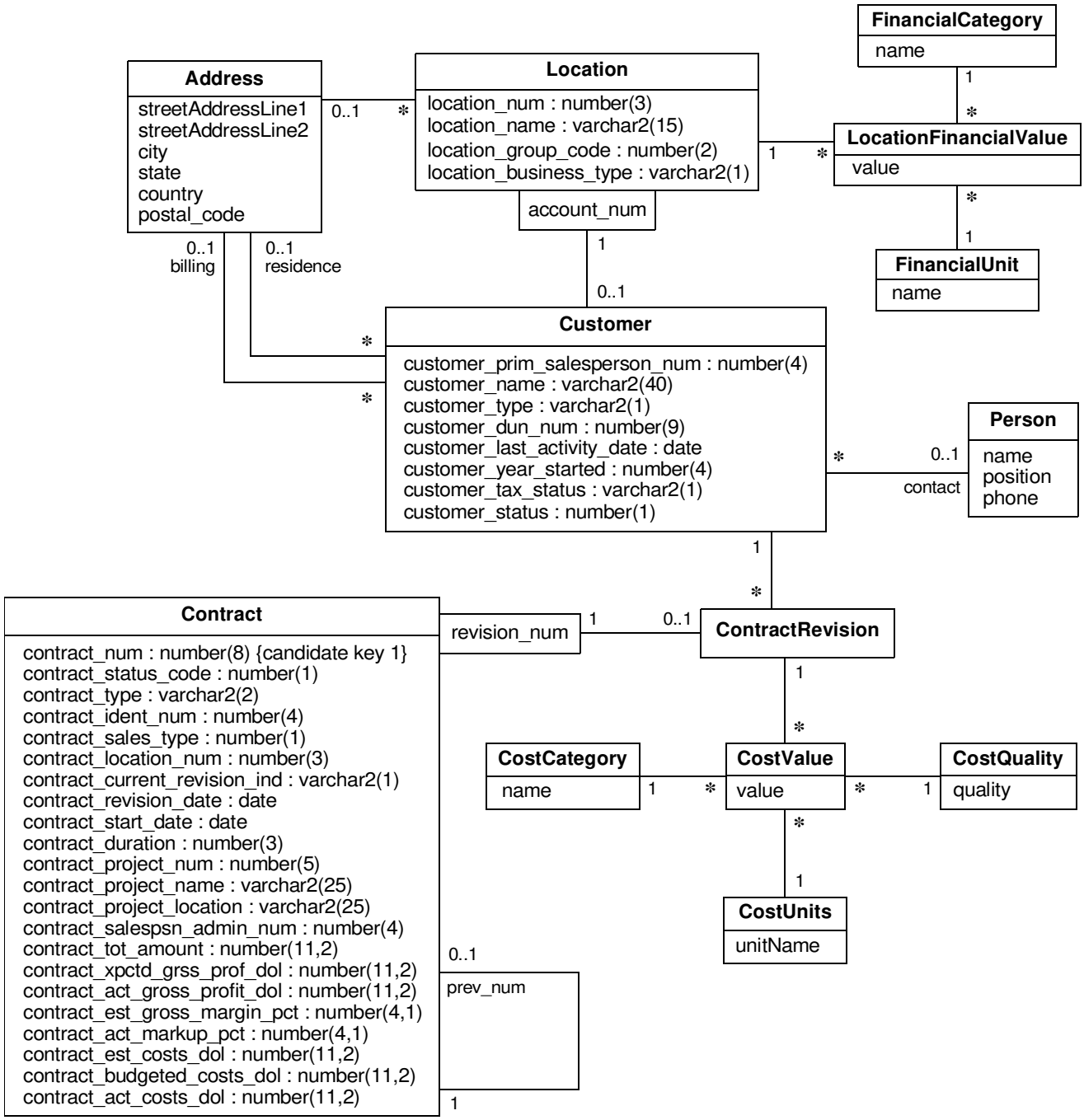

Figure 3 Model after Partial Analysis Recovery

7. Ultimate Disposition of the Project

I originally became involved in the project when a manager for my client company handed me the database structure and asked me to take a look. I suspect he asked me to look at the project because the developers were behind schedule and their code did not yet work. After looking at their database, I was not surprised.

The aftermath of reverse engineering was frustrating—the developers did not care to listen and did not understand many of my criticisms. They did not have the foggiest idea about how to design a database. In fact I was unable to finish analysis recovery because they stopped answering my questions. I was somewhat grateful because I found it painful to interact with them.

Even though management lacked confidence, the developers thought they were doing a fine job and just wanted to keep hacking away. The project eventually ended when management cut their budget. In retrospect I think the business justification for the project was marginal and that is why the failure of the project did not seem to cause much recriminations. I had no further interaction with these developers after the project. (I also did not try to seek them out.)

My interaction with the developers of this project is not typical of my reverse engineering experiences. For most projects the developers have been willing to listen—either because they are proficient and know it or because they realize they need help. In all cases where funding held and the developers were cooperative, I have been able to help them fix their design and get their project on track. Often I have trouble communicating fine aspects of database design, but most developers cooperate as an act of faith because they follow some of what I explain and have seen the benefits.

8. Related Work

This paper is purely an experience report. It does not directly advance the state of the art. The approach is not novel; rather, it is typical of the work of a skilled database reverse engineer. Nevertheless, the reverse engineering community does need clearly documented examples—to document reality and the chaos that errors cause.

The references listed at the end of the paper provide context for database reverse engineering.

9. Conclusions

This paper has only considered database structure. I told you a little about the application gleaned from my discussions with developers, but there was no other input. Even so, we were able to learn much about the application. For a skilled database reverse engineer, the analysis presented in this paper would take about a day of work.

This paper is representative of the kind of insight you can obtain with database reverse engineering. For information systems, once you understand the database, you have much insight into the software as a whole.

10. References

[1] Peter H. Aiken. *Data Reverse Engineering*. McGraw-Hill, New York, New York, 1996.

[2] C Batini, S Ceri, and SB Navathe. *Conceptual Database Design*. Benjamin/Cummings, Redwood City, California, 1992.

[3] Michael Blaha. On Reverse Engineering of Vendor Databases. *Fifth Working Conference on Reverse Engineering*, October 1998, Honolulu, Hawaii, 183–190.

[4] Kathi Hogshead Davis. August-II: A tool for step-by-step data model reverse engineering. *Second Working Conference on Reverse Engineering*, July 1995, Toronto, Ontario, 146–154.

[5] JL Hainaut, V Englebert, J Henrard, JM Hick, and D Roland. Database evolution: the DB-MAIN approach. *13th Entity-Relationship Conference*, Manchester, UK, 1994, 112–131.

[6] Andrew McAllister. Reverse Engineering a Medical Database. *Third Working Conference on Reverse Engineering*, November 1996, Monterey, California, 1996, 121–130.

[7] N Mfourga. Extracting Entity-Relationship Schemas from Relational Databases: A Form-Driven Approach. *Fourth Working Conference on Reverse Engineering*, October 1997, Amsterdam, The Netherlands, 184–193.