Method and Process for Iterative Reengineering of Data in a Legacy System

Alessandro Bianchi, Danilo Caivano, Giuseppe Visaggio
Dipartimento di Informatica – Università di Bari
Via Orabona, 4, 70126 Bari - Italy
bianchi@di.uniba.it caivanodanilo@libero.it, visaggio@di.uniba.it

Abstract

This paper presents an iterative approach to database reengineering, starting from the assumption that for the user organization, the data are the most important asset in a legacy system. The most innovative feature of the proposed approach, in comparison with other rival approaches, is that it can eliminate all the ageing symptoms of the legacy data base. The new database can therefore be readily used to integrate data used by new functions introduced in the legacy software. Moreover, the approach allows all the services offered by modern data base management systems to be exploited. To test the effectiveness of the process described in this paper, it was experimented on a real legacy system; the results reported in the paper confirm its effectiveness.

1. Introduction

In the last few years, there has been a revival of interest in legacy systems. In practice, currently working legacy systems are such important commodities that both the scientific and industrial communities and practitioners have had to pay attention to the problems arising in the most commonly used programming language in the business application domain. They all agree that COBOL will continue to be a major programming language for business applications over the next 10 years [CK00], [Coy00a]. It is estimated that there are over 100 billion lines of code in legacy systems [Coy00b], and several dedicated tools are at present conquering the reengineering tools market [Arr00].

However, some aging symptoms that often affect legacy systems require reengineering of the systems themselves, to avoid the need to discard the whole system. As legacy systems contain key organizational knowledge acquired during the lifetime of the organization, understanding the system is a critical factor for the organization's success. A renewal process can update the legacy system and preserve the organization's investment.

The reengineering techniques available in literature [Big89], [Bro93], [CC90] institute a reengineering process that involves the entire software system. For this reason, the software system must be frozen until execution of the process has been completed; in other words, no changes are possible during this period. In fact, if a change or a system evolution were introduced, the legacy system and the renewal candidate would be incompatible, and the software engineers would have to start the process all over again from the beginning. This situation causes a loop between the maintenance process and the reengineering process.

To overcome this problem, several authors have suggested wrapping the legacy system, and considering it as a black-box component to be reengineered. In this way, it is possible to add new functions and a new, more useful interface with relative ease [Sne96], [WBS97], [DWS00]. A good state of the art analysis is made in [Coy00b] and [BLW99]. However, this technique does not solve the problem of inertia of the legacy system, which will remain unchanged. For this reason, if the wrapped system needs to be evolved in some way, all the consequences of the aging symptoms indicated by [Vis97] will re-emerge. Therefore, although the wrapping approach offers a solution to the problem of coexistence between the aged programs and the new ones requiring relatively little effort, it does nothing to solve the problem of maintenance of aged programs.

Very often the legacy system is so big and offers functions with so critical an importance to the application domain that it would be very risky to replace it with a new system. In fact, as replacement is very costly and is distributed over a very long period of time, the risks involved in developing a new system are often too high. The authors propose iterative reengineering of the legacy system as an alternative. At each iteration, the increment is applied to components of different size but the legacy system can proceed with its usual operations during the reengineering process. Moreover, the legacy system can be wrapped if the legacy administrator judges this to be necessary. However, this eventuality is not included in the considerations presented below.

The main benefit of iterative reengineering is that only one component is frozen each time: as the time spent to reengineer a single component is relatively short, iterative reengineering has the great advantage of cutting down the

Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)
1095-1350/00 $10.00 © 2000 IEEE

time during which the component will necessarily be inert. In practice, reengineering stable components does not have to be a rapid process because these components are unlikely to need maintenance, whereas reengineering non-stable components must be done very quickly, as they generally require frequent maintenance. The best answer is to reduce the size of unstable components before reengineering them.

Another problem in reengineering aged data-oriented legacy systems is that they present many aging symptoms concentrated in the database structure [Vis97], so it will be necessary to improve the database structure to enable the target structure to overcome these symptoms.

The method proposed herein enables improvement of the database structure regardless of the application functions using it, so that both the aged and the target database can coexist and be used by the target reengineered programs and by new, added programs. As reengineering proceeds, the legacy programs and data iteratively decrease while the reengineered programs and data increase.

Since the legacy systems used to conduct the experimentation were data-centered, the reengineering process was executed by the authors according to the following priorities:

- data reengineering;
- procedure reengineering.

In this paper, only the data reengineering process, which is the most important, is presented.

The paper is organized as follows: firstly our approach is compared to the two most important rival approaches (section 2); then the architectural components needed to realize the approach are delineated (section 3). We go on to describe the reengineering process to be executed according to this approach (section 4). The next section describes the ageing symptoms which can be overcome with the proposed approach, and how they are solved (section 5). The experiment is then reported, together with its results and the relative comments (section 6). Finally, some conclusions are drawn (section 7).

2. Related works

The most similar approaches to the one described here are the *Chicken Little Strategy* [BS95] and the *Butterfly Methodology* [WLB97].

These approaches, like the one proposed in the present paper, are based on the assumption that *the data in a legacy system are logically the most important part of the system and that from the viewpoint of the target system development, it is not the ever-changing legacy data that is crucial, but rather its semantics or schema(s)* [WLB97]. Like the other two, the present approach enables reengineering of the data through successive iterations.

The Chicken Little strategy allows the legacy system to interact with the target system during migration, using a mediating module generally known as a Gateway. The Chicken Little uses a forward gateway to translate and redirect calls to the target database service and to translate the target database results to be used by the legacy code. A reverse gateway maps the target data to the legacy database. This solution requires legacy data to be duplicated and reengineered in the target system: for each data access, therefore, both data bases need to be accessed. However, many of the complex features found in modern databases, such as integrity, consistency constraints and triggers may not exist in the legacy database and hence cannot be exploited by new applications [BLW 99], so it is hard to update the system's consistency. The method proposed in this paper translates legacy data into new data stored in the new database without duplicating them during migration. The knowledge of the physical location of each datum, i.e. if it has to be retrieved from the aged or the new data base, is concentrated in only one component called the *Data Locator*, described in the next section. Because the data in the database are unique, the problem of coexistence among data duplicated while reengineering does not exist.

In the Butterfly Methodology, both the legacy and the target data system can continue to operate, but without sharing functions or data.

The Butterfly methodology thus focuses on legacy data migration and develops the target system as an entirely separate process [BLW99]. The old data migrate to the new database, then the old database is frozen and used only for reading. All changes are kept in a temporary auxiliary store (TS). Therefore, each time the system has to access some data it has to read both databases (the old and the target one), as well as the TS, to verify whether the data have yet been updated. The risk that may occur in data-oriented systems with a lot of transactions is that the response time may become very protracted and in this case, the TS will tend to increase in size day by day.

When the information system to be rejuvenated needs to evolve data by extending the database, new data are directly structured into the new database. Moreover, new functions are realized in a *Modern System*, which accesses the same database. The Modern System does not need to know exactly where the data are stored: if it uses the legacy system data, the Data Banker will know this and will access the data through the Data Locator.

The advantage of our method is that the data structure is reengineered, rather than simply migrating the data as in the rival approaches. Therefore, the method overcomes the aging symptoms caused by the degraded quality of the data structure, whereas with the rival methods this problem will continue to be reflected in the new database.

3. The iterative Reengineering approach

With an iterative approach, it is necessary to guarantee coexistence of the reengineered and new systems with the legacy system throughout the reengineering process.

In Figure 1, the information system is shown to include a *Legacy System*, and a *Legacy Database*. The legacy system is a working software system which supports some of an organization's business functions. It is one of the organization's important assets and repository of the know-how that has evolved with it. The legacy database consists of the set of data produced / used / modified / consumed by the legacy system. Both the legacy system and the legacy data base are *aged*, in other words their quality and economic returns are decreased so that maintenance has become uneconomic, requiring a higher effort than the corresponding returns.

The *New Database* includes all the data used by the business processes: we shall call these data *essential*. It also contains *Data Keys* for identifying the data structures. Some Data Keys may be meaningless for the application domain but the software engineer needs them to access the database effectively.

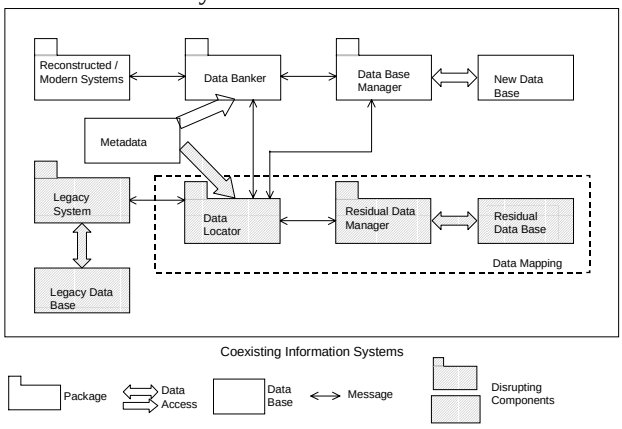


Figure 1. Coexistence of the legacy and reengineered systems during the reengineering process

It is worth noting that the new database is organized according to the target database management system and to the normalization criteria introduced to improve maintenance and database evolution activities. During the reengineering process, it may be necessary to develop new applications in the same application domain. We shall call them *Modern Applications*, because they are located in the same application domain as the legacy system, they share many data with the latter, and they sometimes need to extend the legacy database with new data. Thus the Modern Applications database will be integrated in the New Database. This evolution of the database will be absolutely transparent for both the legacy and the reconstructed systems, because the Data Banker and Data Locator know its physical structure. As a result, after the reengineering process, the New Database will contain the whole database belonging to the current information system, even if this has evolved during the reengineering process. The Reconstructed and Modern Systems access data through service requests to the *Data Banker*. These requests, besides identifying the required service (read, write, delete, …), specify the data the service must operate on: these data are identified through the entities used by the business processes the information system refers to. The *Data Banker* analyzes the structure of the service request, interprets its contents and implements the accesses to the physical database needed to satisfy the service, as well as the physical components containing the data referring to the required service entities. For example, let us consider the service request to open a database and initialize the position of the current record: this is accomplished by the service request open_service. The expression of such a request is shown in Figure 2, where the *italic* strings enclosed in the asterisks are comments.

open_service = **Open a data entity and initialize the position of current instance**
access_mode + entity_name + entity_exist + open_mode + optional_clause

Where:

access_mode = **Specifies the access mode to the entity** [dynamic_access | random_access | sequential_access]

*entity*_name = **Name of the entity**

*entity*_exist = * *Boolean Flag indicating the entity existence before program execution**

open_mode = **Opening mode of the entity** extend_mode | input-output_mode | input_mode | output_mode

Figure 2. An example of a service request

When the reconstructed or newly constructed systems need to open a data base, they deliver the request open_service, specifying the correct values of the parameters indicating the access mode to the entity and its name. Data Banker, using the information in Metadata, associates the entity to be opened with the real tables implementing it in the data base. If all the tables are correctly opened, the flag file_exists is valued as true.

For these reasons, when the database structure changes, the impact to be managed refers to the relationship between entities and database by means of the parameters in Metadata and Data Banker. The application has to be changed only when the entities in the application domain change. Coexistence between the legacy and reengineered data is guaranteed by the *Data Mapping* architecture (enclosed within the dotted line in figure 1) which has several components. The *Residual Database* includes data referring to some components of the legacy database which, although used by the legacy system, are not essential. When the legacy system programs are reengineered they are replaced by new components: these new components will no longer use residual data, so the residual database will be discarded when the reengineering process ends. For example, Table 1 shows some data migrated to the *residual database*. Each datum is described by: the *data name* that identifies the data, the name of the *data file* where the original data were

recorded in the legacy database, the *data description*, to clarify the meaning of the identifier, built using the Italian language. In the excerpt, the data named datfor-arrot-far indicates, for data concerning a pharmacological supplier, that a given amount of a pharmacological product has been ordered so a given discount can be applied. The datum named datfor-arrot-par is analogous, except it concerns non-pharmacological products. The data named datsconto indicates the given discount.

Data Name	Data File	Data Description
datfor-arrot-far	File_0012	Indicates if the amount of a pharmacological product ordered is greater than a threshold
datfor-arrot-par	File_0012	Analogous to the above, but with respect to non-pharmacological products
Datsconto	File_0015	The discount obtained if the amount ordered exceeds the threshold

Table 1. An example of residual data

When the programs in the legacy system need such data, they access the residual database via the *Data Locator*, and can operate as usual. The *Data Locator* is the data mapping component, which translates data from the legacy into the new reengineered format, and vice versa. The legacy system can access new, reengineered data through service requests to the data locator. The data locator behaves towards the legacy system in the same way as the data banker does, with respect to the reconstructed and newly constructed systems. The requests identify the required service and specify the data the service has to operate on; these data are identified through the entities used by the business processes the legacy system refers to. The *Data Locator*, like the Data Banker, analyzes the structure of the service request, interprets its contents, and implements the accesses to the physical database needed to satisfy the service, as well as the physical components containing the data referring to the required service entities. The *Metadata*, represented between the *data locator* and the data banker, is the database containing all the information on the state of the legacy data files and the mapping between the old data and their new localization in the reengineered database. Table 2 shows an example of a piece of data recorded in the Metadata data base, that identifies each datum by name and indicates all the support information allowing specification of the data in the new database. The first two columns, (id data file and data name) refer to information for recognizing the field in the legacy system: for example the fields described in the first three rows are identified in the file by number 1; the field in the fourth row is identified in the file by number 2; the first datum is named DATFOR-REC and indicates the record of the pharmacological products supplier; the second is DATFOR-KEY, i.e. the primary key by which data in DATFOR-REC are identified, the third is DATFOR-TIPO (the type of supplier, wholesaler, dealer, …), and the fourth is DATFOR-ASSINDE (that indicates that the supplier also belongs to the Italian consortium ASSINDE charged with the management of expired pharmacological products). The column format reports the way the legacy system considers the field. The fields described in the first two rows are considered as a group, the third as an alphanumeric value; other possible values for the format are numeric, signed numeric, etc. The column index is used to map a field when an OCCURS function (in COBOL this enables tables to be defined) is specified in the file description of the data file. The key type column records additional information about the use of the field in the legacy system, such as a key to access the data file. In the previous table, only data in the second row represent the primary key. The legacy start and legacy end columns indicate the starting and ending position of the field in the legacy record where the information to be considered is recorded. The following column, field or function name, may indicate the name of the corresponding field in the reengineered database, or if delimited by the '%' symbol, the name of a data locator function. In the excerpt, the first two rows do not refer to any field in the reengineered database, or function in the data locator; the third row indicates that the field DATFOR-TIPO in the legacy database corresponds to the field datfor-tipo in the reengineered database; the fourth row indicates that the field DATFOR-ASSINDE in the legacy database corresponds to the function of the data locator named VERIFICA CONVENZIONE (i.e. verify the convention with the Italian consortium).

Id Data File	Data Name	Format	Index	Key Type	Legacy Start	Legacy End	Field or Function Name	Data Type	Start	End	Owner	Foreign Table	Primary Key	Path
1	DATFOR-REC	group			1	1224	% Nothing to do%				System			
1	DATFOR-KEY	group		primary key	1	7	% Nothing to do				System			
1	DATFOR-TIPO	alpha numeric			342	342	Datfor tipo	String	1	1	Residual data		false	
1	DATFOR-ASSINDE	unsigned numeric			343	343	% Verifica convenzione%				System			

Table 2. An excerpt of the Metadata database

The column data type contains information about the corresponding data type in the reengineered data base. The next two columns, start and end, indicate the starting and ending position of the corresponding data in the reengineered database. The column owner indicates the name of the table in the reengineered database containing the field described by field or function name. In some cases (rows 1, 2, and 4), it has the value SYSTEM, which indicates that it is an internal function of the data locator. In some other cases (i.e. row 3), the owner can be another component (the RESIDUAL DATA BASE). The foreign table column establishes the name of the table for which the reengineered database field is a primary key. In the excerpt, this case is not represented. The primary key column contains, for data in the reengineered database, the Boolean value TRUE if the field is a primary key, FALSE otherwise (as in the case represented). Finally, column path lists the tables the Data Locator will navigate to access the field.

Going back to figure 1, we note that the gray background is used to represent components which will be discarded at the end of the reengineering process. The legacy system will be replaced by the reconstructed and newly constructed systems and the legacy database will be replaced by the new database; the residual database will be progressively emptied as the reengineering process proceeds, while on completion of the process, the usefulness of the *data locator* will also end.

4. Summary of the reverse engineering process

The proposed iterative reengineering process involves breaking the system down into different parts, for each of which the process described below is executed:

1. *Analyze* the current archives;
2. *Redesign* the data;
3. *Adapt the* legacy programs to the new database;
4. *Migrate* data from legacy system to the new database;
5. *Equivalence Test*;
6. *Reworking*;
7. *Iterate* from step 1.

4.1. Analyze the current archives

This phase requires identification, analysis and interpretation of the legacy system data and identification of their mutual relationships. In accordance with [Vis97], four types of data have been identified:

- *conceptual data*, i.e. data which are specific to the application domain and describe specific application concepts. Their meaning is easily understood by the application users but they are quite difficult for the reengineer to understand: a good knowledge of the domain is required to interpret their precise meaning correctly;
- *control data*, i.e. data which are created by a program decision or record the events occurring in the real world or in the system;
- *structural data*, i.e. data used to organize and support the data structures of the system;
- *calculated data*, i.e. data resulting from the application of calculations.

During this phase, possible synonyms of data have to be discovered. Synonyms are then renamed and duplicated data are removed. The conceptual data will be organized in the new database in the next steps of the process; control, structural and calculated data will be merged in the residual database.

Data Name	Data Type	Description
Datfor-anno-autorizza	Conceptual	Year when the legal authorization was obtained
Datfor-anno-revoca	Conceptual	Year when the legal authorization was revoked
Datfor-arrot-far	Control	See description in table 1
Datfor-arrot-par	Control	See description in table 1
Datfor-assinde	Control	See description in table 2
datfor-avere-euro	Calculated	Total amount (in euro) due to the pharmacological supplier: this is calculated starting from the amount expressed in Italian Lira
datfor-banca	Structural	Foreign key used to access Banks data
datfor-base	Control	Flag indicating the official Register the pharmacological product is recorded in
arc-cod-nome	Conceptual	Name of the pharmacological product recorded in the archive of all products managed by the official register named CODIFA.
arc-fed-nome	Conceptual	Name of the pharmacological product recorded in the archive of all products managed by the official register named FEDERFARMA.

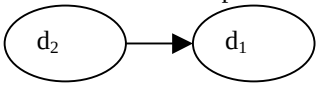
Table 3. An excerpt of classification of the data in the legacy database

Table 3 reports an excerpt of the table obtained after classification of the data in the legacy database. Here, too, the names are in Italian, so for the sake of clarity, the description column has been added.

4.2. Redesign the data

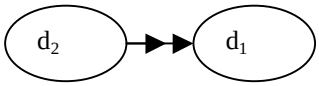
This phase involves redesigning the data organization, identifying the dependencies among them. In general, as shown in [Smi85], assuming that d_1 and d_2 are either single or composite data fields, then:

- there is a *single valued* dependency between d_2 and d_1 if, for each value for d_2, one and only one value of d_1 is retrieved. This is represented by



where d_2 is a "Key" in that each value of $d_2$must be non-null and unique, and determine the value of d_1.

- there is a *multiple valued* dependency between d_2 and d_1 if, for each value for d_2, one or more values of d_1 is / are retrieved. This is represented by



where d_2 and d_1 are "Keys" in that both must be non-null and unique, and each combination of d_2 and d_1 must be unique. Furthermore, each value of d_2 determines a set of values of d_1.

Some functions require determination of the value of d_1 starting from the value of d_2 (i.e. $d_2 \rightarrow d_1$), while other functions require determination of the value of d_2 starting from the value of d_1 (i.e. $d_1 \rightarrow d_2$). In this case, the software engineer must select the dependency required by the functions most critical for the effectiveness of the system. This will cause the database to be structured according to this decision, and will be an effective solution for all the functions requiring that type of dependency. The dependencies among all data to be included in the database are depicted in the *dependency diagram*. Starting from the dependency diagram, tables in Fifth Normal Form are obtained with the technique described in [Smi85]. During the normalization process, the software engineer may need to insert key data which do not belong to the set of conceptual data. In this case, if possible, the structural data found in the legacy database will be reused: they will be removed from the residual database and merged in the new database. Since the adopted approach is iterative, the dependency diagram referring to the part of the legacy system under reconstruction must be integrated with the dependency diagram of the reconstructed parts. During this phase, while defining the access mode to the new database, the software engineer also defines the data to be organized in the Metadata database. Finally, this phase allows any existing redundancies to be observed and redundant data to be eliminated.

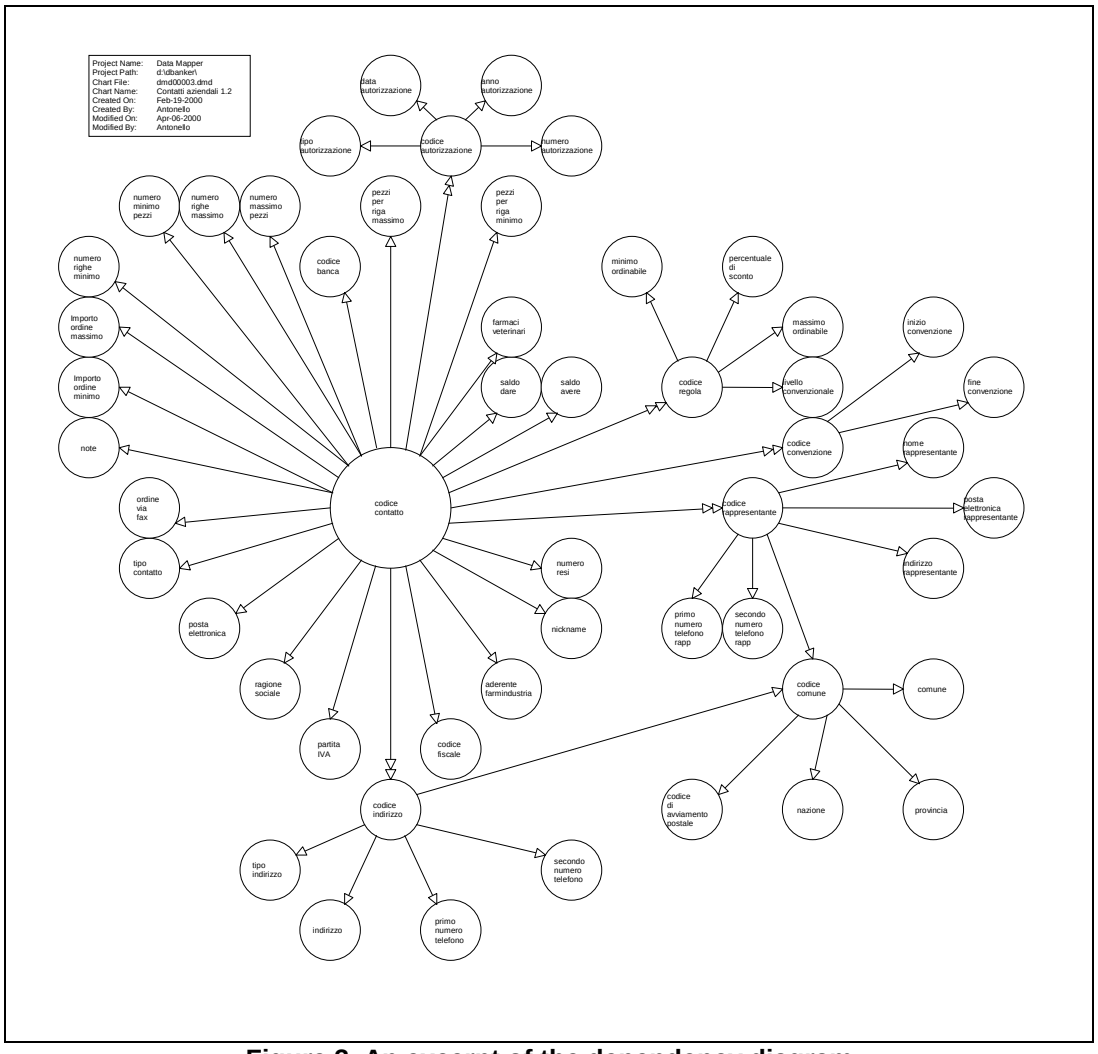


Figure 3. An excerpt of the dependency diagram

For example, Figures 3 and 4 show an excerpt of the dependency diagram obtained during this phase, and the corresponding normalized tables with the navigation paths. In Figure 3, there are a number of bubbles, each of which represents a data field; in each bubble there is a label corresponding to the name of the data field. For example, the large central bubble, labeled "codice contatto" (i.e. the code identifying the pharmacological firm in Italian) identifies the field with the same name. A bubble can have one or more arrows pointing to it and/or one or more arrows pointing out. An arrow pointing from bubble B_1 to bubble B_2 indicates that there is a logical dependency between the fields corresponding to the two bubbles: if F_i is the field corresponding to bubble B_i, the arrow indicates that F_2 is localized starting from F_1. On the basis of the previous definitions, the arrows can be single- or double-headed, indicating a single-valued dependency (i.e. each instance in field F_1 determines an instance in field F_2) or a multi-valued dependency (i.e. each instance in field F_1 determines a set of instances in field F_2). In Figure 3, a number of arrows point out from the large central bubble "codice contatto": among them, the single-headed arrow pointing towards bubble "numero resi" (i.e. the number of expired products returned to the Italian Ass. Inde) indicates that an instance in the field "codice contatto" identifies a unique instance in the field "numero resi". On the other side, the double-headed arrow pointing from the bubble "codice contatto" to the bubble "codice rappresentante" (i.e. the code of the Italian pharmacological sales representative) indicates that the same instance in the former identifies a set of instances in the latter.

In Figure 4 there are a number of linked tables: each table groups together a set of one or more fields corresponding to the bubbles in the previous chart, according to Smith's algorithm [Smi85]. Each arrow pointing from table T_1 to T_2 indicates a path for reaching a seek field in T_2 starting from a given field (foreign key) in T_1. For example, the arrow pointing from the field "codice contatto", that is a foreign key of the table "CONVENZIONI" (the table listing the conventions established between the pharmacological firm and the Italian National Health Service), to the table "CONTATTI" (the pharmacological products supplier) allows identification of the unique record in this last table having the field "codice contatto" equal to the starting foreign key field.

4.3. Adapt the legacy programs

Each program in the legacy system which accesses data which have been reengineered during the previous steps has to be adapted to be able to access the new database.

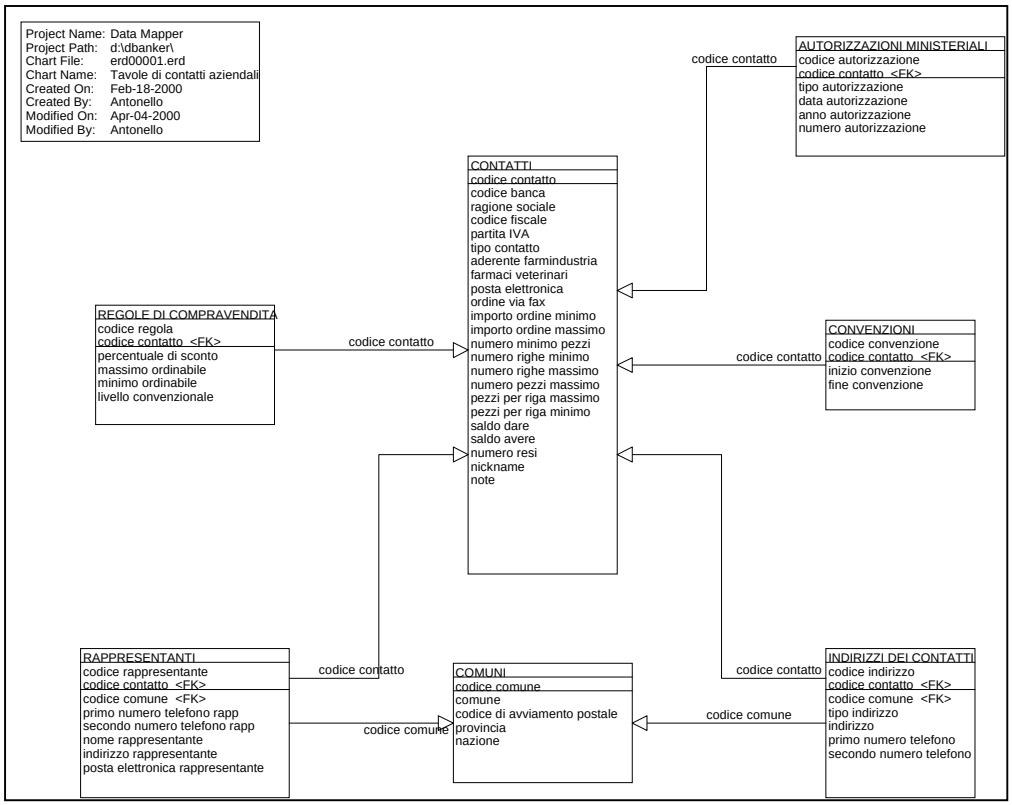


Figure 4. An excerpt of the tables and path derived from the diagram in figure 3

This implies that each of these programs has to be analyzed to identify the instructions accessing data. These instructions will be replaced by service requests to the *data locator*.

Legacy Program

```
:
:
LETTURA-DATFOR-DIRETTA.
    READ DATFOR INVALID GO PRENDI-DITTA-ALFABETICA.
    IF LCK-R            GO LETTURA-DATFOR-DIRETTA.
    MOVE X4             TO W4.
    GO DITTA-SELEZIONATA.
:
```

Adapted Program

```
:
:
LETTURA-DATFOR-DIRETTA.
    CALL "READ" ID-DATFOR DATFOR LEGACY-FILE-STATUS.
    EVALUATE LEGACY-FILE-STATUS
        WHEN 00 GO PRENDI-DITTA-ALFABETICA.
        WHEN 99 GO LETTURA-DATFOR-DIRETTA.
    END-EVALUATE
    MOVE X4             TO W4.
    GO DITTA-SELEZIONATA.
:
```

Figure 5. An excerpt of code before (above) and after (below) adaptation

Figure 5 shows an example of adaptation of the legacy program: above, there is the piece of code belonging to the legacy system program and below, there is the corresponding piece of code after adaptation. More precisely, the piece of code in the legacy system includes a label ("LETTURA-DATFOR-DIRETTA.", meaning direct access in reading mode to the DATFOR data) and a reading instruction which, if this fails, jumps to the label "PRENDI-DITTA-ALFABETICA." (i.e., get necessary data by reading from the alphabetical list); if the reading instruction succeeds, it proceeds with the next IF-instruction. If the record to be read is currently locked by some other process, then it loops and waits for the record to be unlocked. When the record is unlocked, it copies the contents of variable X4 to the variable W4. Finally there is a jump to the label "DITTA-SELEZIONATA." (i.e. selected firm). After adaptation, the previous piece of code continues to be identified by the same label ("LETTURA-DATFOR-DIRETTA."), but the reading instruction is replaced by a call to a new program named READ, and passage of the parameter ID-DATFOR (the identifier of the firm), DATFOR (the file to be read), LEGACY-FILE-STATUS (a return parameter indicating the current status of the legacy file). After reading, the return parameter is evaluated: if it is 00 (i.e. the reading failed), it jumps to the label "PRENDI-DITTA-ALFABETICA.", if it is 99 (i.e. the record was locked), it jumps to the label "LETTURA-DATFOR-DIRETTA.", otherwise it proceeds with the instruction which copies the contents of variable X4 to the variable W4. Finally there is a jump to the label "DITTA-SELEZIONATA."

4.4. Migrate the data

Data in the legacy database are progressively migrated to the new database; instead, data identified as not essential are migrated to the residual database.

4.5. Equivalence test

The information system obtained with the previous steps is now ready to run, offering the same functions as the legacy system. To verify this, an equivalence test is executed comparing the behavior of the legacy system with that of the reengineered system.

4.6. Reworking

For each failure identified during the equivalence test, the faults generating it must be analyzed and then the reengineered system modified to eliminate the failure. After this modification, it is necessary to execute a regression test aiming to validate the equivalence. The loop will stop when no further failure is identified.

4.7. Iteration

Finally, another part of the legacy system to be reengineered is identified and the process starts again from step 1. During the iteration, the software engineer may realize that, since all the sets of data operated on by one or more programs have been migrated, these programs can be reconstructed. It is also worth iteratively reengineering the procedures, to cut the time required for application of the process to each segment in the legacy system. Reengineering of the procedures is not considered in this paper.

5. Aging symptoms overcome

The process proposed overcomes some of the ageing symptoms of the legacy system. With reference to the classification discussed in [Vis97], the data reengineering method described in this paper aims to eliminate the following ageing symptoms in the legacy database: *Data Pollution*, *Temporary Files* and *Pathological Files.*

Data pollution occurs when the data base stores redundant data and data that are not essential for the application domain. Redundancy can be caused by:

- *semantically redundant data*, i.e. if the definition domains of two or more data are the same or are contained one in the other, or each equal value in the two definition domains can be interpreted in the same way by both data; *computationally redundant data*, i.e. data which can be computed starting from a set of other data included in the same database;

- *structural data* i.e. data used to support the architecture of the legacy database, which have been added to the basic architecture when evolutions of the legacy system did not enable modifications with the original architecture;
- *control data* i.e. data used to communicate to one or more components of the legacy system that some events occurred during the execution of other components.

Data Pollution causes the maintainer to encounter increased difficulty in understanding the legacy system and database, and increases the impact of many modifications, making maintenance activities unreliable.

For example, in the case study legacy system, the identifiers of the pharmacological products are stored in different files, with different names: all of these are semantically redundant. A solution to this problem is to record all this information in a unique field in the new system. Referring to table 3, the last two rows indicate the fields in two different archives where the names of the pharmacological products are recorded: many products are managed both by the official Register CODIFA and by the official Register FEDERFARMA, so the names of many products are duplicated in the two files.

In the same system, there were many examples of computationally redundant data, obtained on the basis of the essential data already stored in the data base (see table 3). An example of structural data is the datum named datfor-banca. This is a key allowing identification of the table where the supplier's banks are recorded.

Finally, some examples of redundant control data shown in Table 3 are datfor-arrot-far, datfor-arrot-par, datfor-assinde, and datfor-base, described above.

All data causing pollution will be removed from the New Database. However, while the legacy system is still running, they are managed in the Residual Database, to allow legacy programs to update and use them. In the new system, it might even be useful, for the sake of effectiveness, to introduce some redundancies, especially computational redundancies. In this case, the Essential Database should be extended with the redundant data, and this will have an impact on the Metadata Database and the Database Manager components. These redundancies may make maintenance more difficult but could be justified by the need to increase the effectiveness of the system.

Temporary Files are files created and read by the system but not updated. These files indicate an aging symptom which has onset for the following two reasons [VIS 97]:

- a program Pr_i generates a datum d_{ij}, which is live in S, for a time decided by its designer. After this decision has been taken, the application is extended by one or more programs Pr_1, …, Pr_k, which need to use the same datum even after it has been destroyed by Pr_i. To solve the problem, a file F_i is created and d_{ij} from Pr_j is stored in it before the latter is destroyed. Sometimes Pr_j is the very program which will kill d_{ij}.
- a program Pr_i creates data in F_k which will be useful to the other programs Pr_1, …, Pr_k which have no access to F_k. Owing to the complexity of F_k, rather than modifying all these other programs to give them access to the latter, the maintainer decides to create a temporary file F_i, in which Pr_j memorizes the data extracted from F_k and to modify programs Pr_1, …, Pr_k to give them access to F_i. In this case, too, Pr_j and Pr_i will sometimes coincide.

The elimination of *case a* is guaranteed in the proposed approach by the adoption of temporal coordinates, which include a temporal dimension in the management of data. When the legacy database is reengineered, the temporal dimension expressed by these coordinates is then managed, as well as the other services offered by the data banker. The solution of *case b* is assured in our architecture by localizing centralized knowledge about the database structure in only one component: the Database Manager.

P*athological files* are those which are created and modified by many programs in the legacy system. Our approach does not overcome this symptom but it leads indirectly towards its solution. In fact, in our architecture the only component that belongs to the whole database is the Database Manager: it verifies that all the operations the programs need to execute on the database preserve the internal consistency of the database itself.

6. Experimental data

The proposed approach has been experimented in the Software Engineering Research Laboratory (SER_Lab) of the University of Bari on an aged legacy software system. The system supports the management of pharmacies, storing information about products sold, chemical and pharmacological issues and legal and economic aspects.

It is used in about 100 pharmacies, mainly located in Southern Italy. The development of this system began in 1987 and the first release was delivered in 1989. It has always been maintained over the years by the same software house, which develops, distributes and maintains it. Currently it is characterized by the following features:

- 600 KLOC;
- 2300 modules;
- 8000 data recorded in the database;
- 350 files managed.

Due to continually changing Italian health service laws, the system has required several adaptive maintenance activities over the years, as the features of products sold by pharmacies have a very high rate of change.

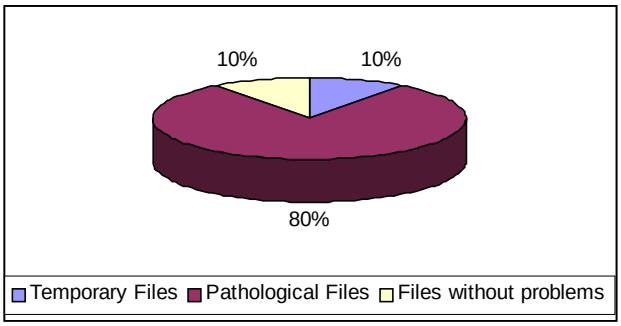


Figure 6. Summary of features of the files in the legacy system

Because of the high number of maintenance activities, the system suffers from several aging symptoms. In particular, referring to the symptoms analyzed in previous section, the legacy presents the following features, graphically summarized in figure 6:

- 10% of files are temporary;
- 80% of files are pathological;
- only 10% of files have no problem.

After reengineering, all the files were problem-free and the first symptoms were overcome.

After the step involving analysis and classification of the current archives, the legacy system presented the features summarized in table 4.

Class of data	% before reengin.	Data before reengin.	% after reengin.	Data after reengin.
Semantic. redundant	10%	800	0%	0
Computat. redundant	6%	480	0%	0
Conceptual	61 %	4880	98%	6200
Control	9 %	720	2 %	126
Structural	8 %	640	0 %	0
Calculated	6 %	480	0 %	0
Total	100%	8000	100%	6326

Table 4. Summary of data classification

Data classification after reengineering was executed on the basis of the contents of the New Database; we can see that the aging symptoms of the data have been removed, too. It is worth noting that the New Database also has a lower total number of data, although the number of essential data has increased in comparison with the legacy database, because some data have been added to realize the temporal coordinates required to replace the removed temporary files.

Finally, the number of control data has become very low: it comprises only the control data used to identify records in the New Database; it should be borne in mind that some essential data are also used as primary keys.

Since the system is mainly data-oriented and featured many of the typical aging symptoms, it was a suitable candidate for experimenting the proposed approach to iterative data reengineering. The legacy database was partitioned into 20 parts, each corresponding to a set of files, and these were iteratively processed. Each iteration required about the same length of time, so only the global data are reported. The effort expended to complete the reengineering process described above was about 2.5 person/years; table 5 details the distribution of effort over the phases of the process.

Phase	Effort (person/days)	Percentage
Analyze current archives	85	17%
Redesign data	145	29%
Adapt legacy programs	90	18%
Migrate data	25	5%
Equivalence tests	75	15%
Reworking	80	16%
TOTAL	500	100%

Table 5. Summary of effort spent on each phase

The “analyze current archives” phase is particularly costly because the software engineers executing the case study had first to understand the meaning and the role of each datum, with little reliable documentation to help them, and often had to read the program source code.

The “redesign data” phase also required an understanding of the programs. Thanks to this, the software engineers were able to define the dependencies among data. The equivalence test required relatively little effort, for two main reasons: the legacy system test cases selected were among the most frequent transactions used by the organization. Moreover, the testing was not done in great depth, because the goal of the experimentation was to assess the method’s ability to treat aging symptoms rather than the equivalence between the reengineered systems and the legacy system.

Finally, we must point out that the software engineers who executed the reengineering, besides understanding the programs through their reading, also referred to the knowledge possessed by the legacy system administrator and maintainer.

7. Conclusions

This paper describes an approach to data reengineering based on the key aspect of iteration, in that it aims to reengineer few parts of the legacy system at a time. As reengineering a single part takes little time, this feature cuts down the time during which the ordinary maintenance activities are frozen.

Secondly, the proposed architecture allows the legacy system to access both the reengineered and the legacy databases. In this way, only a few modifications to the source code of the legacy system are required to allow the legacy system to use the reengineered data. Moreover, the solution permits the legacy system to be endowed with new programs which can directly access the new reengineered data, besides their own new data.

The strengths of the proposed approach are:

- the iterative approach to executing reengineering;

- coexistence between the new and old legacy systems;
- a unique new database which allows new programs to be developed in the same application domain, reusing data migrated from the legacy database to the target database;
- exploitation of all the services available in the target database management system;
- elimination of all the aging symptoms of the legacy database.

The weaknesses of the approach are:

- the need to build and maintain the data locator, which is then removed at the end of the reengineering process; this weakness is minimized by reusing the programs included in the data banker;
- the need to manage the residual database, which also has to be removed after completion of the reengineering; however, at least the approach makes this management transparent for the system maintainer.

An experiment was carried out with the aim of evaluating the overall effectiveness of the proposed method: note that the experimentation did not aim to assess the effort required for reengineering. One of our future works will aim to carry out a similar experiment on an enlarged software system.

Moreover, the experimentation confirms that if the documentation is lacking, a lot of effort in the reengineering process must be spent on reading and understanding the programs. It will therefore be necessary to rely on one or more experts of the application who can aid this comprehension.

Acknowledgments

We would like to thank all the students who participated in the case study for their fruitful work and interesting suggestions, and in particular Dott. A. Genuario and Miss T. Baldassarre for their patient work. We are also very thankful for the diligent and efficacious contribution made by Dott. R. Kudlicka, administrator and maintainer of the legacy system, which greatly aided understanding of the application. Finally, a special thank to Ms. Mary V. Pragnell, B.A. for her contribution as technical-writer

References

[Arr00] E.C. Arranga, "Cobol Tools: Overview and Taxonomy", *IEEE Software*, Vol. 17 No. 2, Mar/Ap 2000, IEEE Comp. Soc. Press, pp. 59-69.

[Big89] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer*, July 1989.

[BLW99] J. Bisbal, D. Lawless, B. Wu, J. Grimson "Legacy Information Systems: Issues and Directions", *IEEE Software*, Vol. 16 No. 5, Sept/Oct 1999, IEEE Comp Soc. Press, pp. 103-111.

[Bro93] A.J. Brown, "Specification and Reverse Engineering", *Software Maintenance Research and Practice*, J. Wiley & Sons, Vol.5, 1993, pp 147-153.

[BS95] M. Brodie, M. Stonebraker, *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*, Morgan Kaufman Publishers Inc., San Francisco, 1995.

[CC90] E.J. Chifosky, J.H. Cross II, "Reverse Engineering and Design Recovery: a Taxonomy", *IEEE Software*, Jan. 1990

[Coy00a] F.P. Coyle "Does Cobol Exist?", *IEEE Software*, Vol. 17 No. 2, Mar/Apr 2000, IEEE Comp Soc. Press, pp. 22-36.

[Coy00b] F.P. Coyle "Legacy Integration Changing Perspectives", *IEEE Software*, Vol. 17 No. 2, March/April 2000, IEEE Computer Soc. Press, pp. 37-41.

[CK00] D. Carr, R.J. Kizior, "The Case for Continued Cobol Education", *IEEE Software*, Vol. 17 No. 2, Mar/Apr 2000, IEEE Comp. Soc. Press, pp. 33-36.

[DWS00] S.C. Dorda, K. Walinau, R.C. Seacord, J. Robert, "A Survey of Legacy System Modernization Approaches" *(CMU/SEI-2000-TN-003) Software Engineering Institute, Carnegie Mellon University*, Apr 2000.

[Smi85] H.C. Smith, "Database Design: Composing Fully Normalized Tables from a Rigorous Dependency Diagram", *Comm. of the ACM*, Vol. 28, No. 8, Aug. 1985, pp. 826-838.

[Sne96] H.M. Sneed, "Encapsulating Legacy Software for Use in Client/Server System" *Proc. Working Conf. On Reverse Engineering (WCRE'96)*, Monterey, Calif., Nov. 8-10, 1996, pp.104-119.

[Vis97] G. Visaggio, "Comprehending the Knowledge Stored in Aged Legacy Systems to Improve their Qualities with a Renewal Process", *ISERN-97-26 International Software Engineering Research Network*, 1997.

[WBS97] N.H. Weiderman,J.K. Bergey, D.B. Smith, S.R. Tilley "Approaches to Legacy System Evolution" *(CMU/SEI-97-TR-014), Software Engineering Institute, Carnegie Mellon University*, Dec. 1997.

[WLB97] B. Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, D. O'Sullivan, "The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information System", *Proc. Int. Conf. Eng. Complex Computer Systems (ICECCS'97), IEEE Comp Soc. Press*, Los Alamos, Calif., 1997, pp. 200-205.