Iterative Reengineering of Legacy Systems

Alessandro Bianchi, *Member*, *IEEE Computer Society*,
Danilo Caivano, *Member*, *IEEE Computer Society*, Vittorio Marengo, and
Giuseppe Visaggio, *Member*, *IEEE*

Abstract—During its life, a legacy system is subjected to many maintenance activities, which cause degradation of the quality of the system: When this degradation exceeds a critical threshold, the legacy system needs to be reengineered. In order to preserve the asset represented by the legacy system, the familiarity with it gained by the system's maintainers and users, and the continuity of execution of current operations during the reengineering process, the system needs to be reengineered gradually. Moreover, each program needs to be reengineered within a short period of time. The paper proposes a reengineering process model, which is applied to an in-use legacy system to confirm that the process satisfies previous requirements and to measure its effectiveness. The reengineered system replaced the legacy one to the satisfaction of all the stakeholders; the reengineering process also had a satisfactory impact on the quality of the system. Finally, this paper contributes to validate the cause-effect relationship between the reengineering process and overcoming the aging symptoms of a software system.

Index Terms—Reengineering, legacy system rejuvenation.

1 Introduction

The importance of a legacy system for the organization owning it as the "backbone of an organization's information flow and as the main vehicle for consolidating business information" [1] is widely recognized by both scientific and industrial communities. These systems and the data they process are vital assets for the organizations that use them. The organization's evolution through the years requires synchronized evolution of the legacy systems; however, such systems should always provide an adequate quality level, so that they can be easily maintained. Unfortunately, due to degradation, legacy systems very often provide low quality levels, and, as a consequence, their maintenance becomes very costly.

Lehman and Belady empirically prove in [3] that, if no improvement is made, maintenance degrades the software quality and, therefore, its maintainability. In [2], some quality factors, called *aging symptoms*, are identified; each of them is then associated to a set of metrics, which allows its quantification. These symptoms become heavier and heavier to manage as the number of maintenance activities increases, so confirming the principles expressed in [3]. Moreover, in [2], some experimental evidence was derived, which showed that the reengineering process can decrease some aging symptoms. In this work, we experiment a further case of external validity of the cause-effect relation between software systems reengineering and overcoming the aging symptoms described in [2]. For the sake of completeness in addition to improving the quality of the system, the reengineering process should make it possible to introduce new functions and adopt new technologies, in order to ensure efficient management of the information container in the legacy system, as explained by Noffsinger et al. in [4] and by Robertson in [5].

The reengineering process is intrusive because it requires the data and the procedures to be restructured all at the same time. Moreover, according to other authors such as Biggerstaff [6] and Brown [7], for example, the reengineering process must involve the entire system. This should make it necessary to block the system during execution of the process and, of course, all maintenance activities should be interrupted until the process is concluded. In fact, each change would have to be executed both in the legacy and in the reengineered system, and there is a high risk that the renewed system would no longer be equivalent to the legacy at the end of the process; therefore, further corrective maintenance would be required. This situation causes a loop between the maintenance process and the reengineering process.

Obviously, however, the legacy system cannot really stop working during the process, and it will also be necessary to satisfy maintenance requests within a short period of time. For this reason, the reengineering process we propose has to be done iteratively and gradually on few procedures at a time and each operation lasts as short a time as possible, so that only requests for change having an impact on the few procedures currently being reengineered need to be frozen.

Due to the iterative nature of the reengineering process, during its execution the system will include both reengineered and legacy components. Both these components must coexist and cooperate in order to ensure the continuity of the system. Finally, any maintenance activities, if required, have to be carried out on both the reengineered and the legacy components, depending on the procedures

- *A. Bianchi, D. Caivano, and G. Visaggio are with the Dipartimento di Informatica, Università di Bari, Via Orabona, 4, 70126 Bari, Italy. E-mail: {bianchi, caivano, visaggio}@di.uniba.it.*
- *V. Marengo is with the Dipartimento di Statistica, Università di Bari, Via Rosalba, 53, 70124 Bari, Italy. E-mail: vmarengo@dss.uniba.it.*

Manuscript received 15 Mar. 2002; revised 19 July 2002; accepted 10 Sept. 2002.
Recommended for acceptance by G. Canfora and A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117300.

they have an impact on. Our approach to reengineering has been organized in such a way as to satisfy all these requirements.

The novelties of the iterative reengineering process we propose are as follows: The reengineering is gradual, i.e., it is iteratively executed on different components (data and functions) in different phases; during execution of the process there will be coexistence of legacy components, components currently undergoing reengineering, reengineered components, and new components, added to the system to satisfy new functional requests.

The Iterative Reengineering process has two advantages: It guarantees that the system will continue to work even during execution of the process, and it preserves the maintainers' and users' familiarity with the system, thanks to making only small, gradual changes during each iteration.

The proposed method has been experimentally applied to reengineer an aged industrial legacy system, called Fa2000. It is an industrial software system supporting chemistry item distributors. It deals with data referring to the chemical companies; pharmaceutical chemistry aspects of the products; health, economical, and legal issues associated to them. The system continued to be used before, during, and after its reengineering. The examples referred to in the remaining part of the paper illustrate the system's reengineering. Note that, being a system developed and used by Italians, the examples reported are all in Italian: Where necessary, an English translation is provided. Although the case study refers to a legacy system written in COBOL, the proposed method is independent of both the programming language and the software platform.

Our investigation about reengineering faces two aspects. The process model we propose contributes to enlarge the body of knowledge in software engineering. In fact, the proposed model satisfies the reengineering requirements better than other known processes. Moreover, it can be generalized so that it can be applied in various contexts. The paper is also interesting for practitioners because the process model is exhaustively described and can therefore be applied in other cases. Moreover, the costs and benefits of applying the process in a real case are reported.

The paper has been organized as follows: Section 2 analyzes the other main approaches described in the literature and points out the innovative aspects of the approach presented. Section 3 describes the iterative reengineering model upon which our approach is based. The approach is general: Only the examples provided for explaining the concepts depend on the applicative context. Section 4 outlines the case study on which the method was experimented: The aging symptoms before and after reengineering the software are outlined, and the costs required for each phase of the process are evaluated. Finally, in Section 5 the main conclusions are drawn.

2 Related Work

The importance of legacy systems and their need to offer high quality levels is acknowledged by the ample literature on these topics. A great number of techniques and methods have been proposed to face the problem: The works by Blaha ([8] and [9]), Sneed ([10] and [11]), Coyle ([12] and [13]), Quilici [14], Robertson [5], and others ([1], [15]) are just a few examples. An exhaustive discussion of this research is outside the scope of the present work and, therefore, in this section, only the works related to the proposed method are discussed.

The proposed approach considers reengineering as a process that involves the whole system, regardless of the specific platform. For the sake of completeness, in such an approach migration toward modern platforms or programming languages is only one of the aspects dealt with, as the main purpose is that of treating system aging symptoms. Our research, in accordance with the view presented by Chikofsky and Cross [16], defines the reengineering process as analysis and modification of the entire system in order to redevelop it in a new format.

In order to carry out this process, many authors (for example, Sneed [11], [17], Comella-Dorda et al. [18]) propose techniques for wrapping legacy systems: The latter is considered as a black-box, covered by a software layer interfacing with the new functions that are added to the system. The state of art of these wrapping techniques is presented in the works by Bisbal et al. [1] and in Coyle's work [13]. They all point out, however, that this technique does not solve the problem of inertia of the legacy system, which remains unchanged. This problem has been emphasized by Visaggio in [2]: "if the wrapped system needs to be evolved in some way, all the consequences of the aging symptoms will re-emerge." Therefore, although the wrapping approach offers a relatively low effort solution to the problem of coexistence of the aged programs and the new ones, it does nothing to solve the maintenance problem of aged programs.

Similar approaches to the one proposed in this work include the *Chicken Little Strategy* [19] and the *Butterfly Methodology* [20]. We share the assumption that "*data in a legacy system are logically the most important part of the system*" and that "*from the viewpoint of development of the target system, it is not the ever-changing legacy data that is crucial, but rather its semantics or schema(s)*" [20].

The Chicken Little Strategy, Butterfly Methodology, and the Iterative Reengineering method discussed in the following carry out reengineering of the legacy system through successive iterations, by applying the process to a (small) set of components during each iteration. This feature allows all these methods to overcome the problem of having to interrupt the system during the entire reengineering process. In other words, only those maintenance requests that refer to the components currently being reengineered need to be frozen, while the remaining parts of the system can continue to evolve independently of the process.

More precisely, the Chicken Little Strategy gradually rebuilds the legacy system on the target platform using modern tools and technology. During the reengineering process, the legacy and target systems make up a composite system, and the components of both of them access data through *Gateways* built for this purpose. The main difference between Chicken Little and Iterative Reengineering lies in the coexistence of legacy and target data. In fact, the Chicken Little strategy does not implement a specific data

reengineering policy; therefore, during process execution, some data in the legacy database can be duplicated in the target database. This duplication can be due to difficulties in making an adequate separation between data management functions and application code using those data, for example. This data duplication is managed by a forward gateway and a reverse gateway. The former is used to translate and redirect calls to the target database service and to translate the target database results to be used by the legacy code; the latter maps the target data to the legacy database. With this solution, for each data access, both databases need to be accessed. Moreover, 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 [1], so it is hard to maintain the system's consistency.

The Butterfly methodology, on the other hand, focuses on legacy data migration and develops the target system as an entirely separate process. Firstly, 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. 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 temporary store, to verify whether the data have yet been updated. The main weakness of the Butterfly methodology with respect to data migration is the need to freeze the old database and to use it only for reading, while changes are kept in a temporary auxiliary store, so that data access time increases. A data-oriented system most likely frequently accesses data. Therefore, an increase in access time may cause a decrease in performance, as pointed out by Sneed in [21]. Moreover, this methodology does not allow the legacy functions to coexist with the reengineered and newly constructed functions [1].

The Iterative Reengineering method shares some features with both these approaches:

- The Chicken Little strategy and the Iterative Reengineering method include analogous steps of analysis and decomposition of the legacy system, reengineering, and migration of interfaces, applications, and databases.
- Both in the Butterfly Methodology and the Iterative Reengineering method, the legacy and the target data system can continue to operate.

Nevertheless, the Iterative Reengineering method can overcome some of the weaknesses of both its competitors. In brief, with respect to the Chicken Little strategy, in the Iterative Reengineering there is no duplication of data, as the use of a Data Banker created to this end allows legacy and reengineered data to coexist. Moreover, the reengineered data are organized in a target database, which can exploit modern technology even if this is not supported by the legacy system. With respect to the Butterfly Methodology, the Iterative Reengineering method has the advantage that both the legacy and the reengineered systems can share functions and data. The main advantage of our method is that it enables coexistence of the data and functional components in the legacy system and those in the reengineered system.

Moreover, in the Iterative Reengineering method, the data structure is reengineered, rather than simply migrated as in the competitor approaches. In practice, the legacy data are translated into the new database without duplication [22]. The legacy database does not have to be duplicated nor frozen, but is gradually transferred into the new database, which can be based on a more modern database management system. The components of the reengineered system coexist with those of the legacy system and use either the legacy or the new database, depending on where the data to be processed are stored. The components of the legacy system will gradually be reengineered and the old system will finally disappear. By the time the legacy system has been completely reengineered, the old database will also have been completely emptied.

3 Iterative Reengineering

3.1 Background

The rationale of our research is illustrated in some works by the authors, such as [2], [22], and [23]. Here, to ensure full understanding of the concepts used in the rest of the paper, the meaning associated to some key words is defined: For further details, the listed works can be consulted.

In the following, by *system,* we mean a set of programs that manage a set of data arranged in a record in a database, which is physically spread over a set of files. We say that a system is a *legacy* if it is operative and constitutes a useful and essential factor in the organization's business function. A legacy system is *aged* when its quality has decayed. In order to verify the aging of a system, the following *symptoms* were identified in [2]:

- *Pollution*, i.e., the system includes many components which are not necessary to carry out the business functions.
- *Embedded knowledge*, i.e., the knowledge of the application domain and its evolution is spread over the programs and can no longer be derived from the documentation.
- *Poor lexicon*, i.e., the names of components have little lexical meaning or are in any case inconsistent with the meaning of the components they identify.
- *Coupling*, i.e., the programs and their components are linked by an extensive network of data or control flows.
- *Layered architectures*, i.e., the system's architecture consists of several different solutions that can no longer be distinguished; even though the software started out with a high quality basic architecture, the superimposition of these other hacked solutions during maintenance has damaged its quality.

Each one of the previous symptoms has been detailed into a set of metrics that can be measured on the system to be maintained. The value for each metric suggests what operations should be carried out to treat the aging symptom. For the purposes of this work, we only focus on the coupling and layered architectures symptoms. For the coupling symptom, we consider only the number of

pathological files, i.e., files created or modified by more than one program; the number of *control data*. With respect to the layered architectures symptom, we consider only the number of *temporary files*, i.e., files which are created/read but never updated/deleted, the number of *semantically redundant data*, the number of *computationally redundant data*, and the number of *redundant structural data*. For the sake of completeness, note that concepts analogous to Visaggio's *aging symptoms* appear in literature under different names, for example, Blaha and Premerlani call them *idiosyncracies* in [24]. Their comparison is outside the purposes of the present work: We focalize the symptoms presented in [2] for reasons of continuity in our research.

For the purposes of this work, all the data managed by a legacy system will be partitioned into two classes: *primary data* and *residual data.* The first are needed to carry out the application's business functions, the latter are not necessary for carrying out the business functions but are used by the legacy system and must therefore remain in the database until the procedures that use them have been reengineered. Also, the primary data include *conceptual data* and part of the *structural data.*

The *conceptual data* are specific to the application domain and are used to describe particular concepts having to do with that application. The system users understand their meaning because they refer to concepts they are familiar with.

The *structural data* belonging to the class of primary data are those used to organize and support the data structure in the legacy system. They are necessary to correctly and efficaciously access the conceptual data; a typical example is the identifying codes that represent the primary keys in the tables where the conceptual data are organized.

Instead, the *residual data* class contains all the data existing in the legacy database, but which should ultimately be eliminated to improve the software quality. They are classified as:

- *control data* that communicate to one procedure the occurrence of an event during execution of another procedure, thus regulating the behavior of the former procedure;
- *redundant structural data* used to organize and support the data structures of the legacy system, but which are not strictly required; a better database design allows to remove them;
- *semantically redundant data* whose definition domain is the same as, or is contained in, the definition domain of other data, while each equal value in the two definition domains is interpreted in the same way;
- *computationally redundant data* which can be computed starting from a different set of data included in the same database.

The above classification of data managed by legacy systems is summarized in Fig. 1.

In order for the system to be gradually reengineered, the process described in the following must be iteratively applied to different components. In this work, according to UML specification UML 1.3 [25] by component we mean "*a physical replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code or equivalents such as scripts or command files.*" Each component may be in one of the following states during execution of the process:

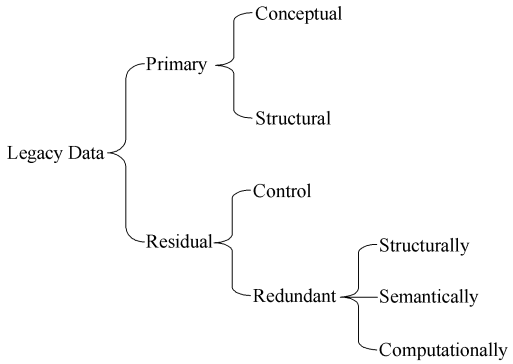

Fig. 1. Classification of data in a legacy system.

- *Legacy*, i.e., components of the legacy system that have not yet been reengineered.
- *Restored*, i.e., functions with the same structure they had in the legacy system, but that access data through the new data banker, that alone knows the physical structure of the database.
- *Reengineered*, i.e., components of the legacy system that have already been modified and whose quality has reached the desired levels.
- *New*, i.e., components that did not exist in the legacy system, but have been added to introduce new functions in the same application domain.

3.2 System Architecture

The iterative reengineering method is based on gradual evolution of a legacy system, by reengineering the legacy system's components in sequence and guaranteeing coexistence among the components, that will go through various different states during the process. Fig. 2 shows the software system architecture while the execution of the reengineering process is going on. The figure shows that a unique architecture encloses both the legacy and the reengineered components. Therefore, the legacy system and the reengineered one coexist while the process is going on.

It is worth noting that restored components are also enclosed in the same architecture, even if only for a short period of time. This architecture allows the software system to be used as usual, even if its components gradually evolve. More precisely, the package labeled as "USER INTERFACE" intercepts the user requests and then activates the corresponding component, that may be in a legacy, a restored, or a reengineered state. Therefore, the systems cooperate in that they share the resources (i.e., data, metadata, and operative environment). All together, they satisfy the users' requests, each providing its own capabilities. Although, while the reengineering goes on, the legacy system's capabilities are migrated to the reengineered system.

All the components labeled as "LEGACY COMPONENTS" in Fig. 2 represent the aged system operating on a set of data recorded in the database labeled as "LEGACY DB"

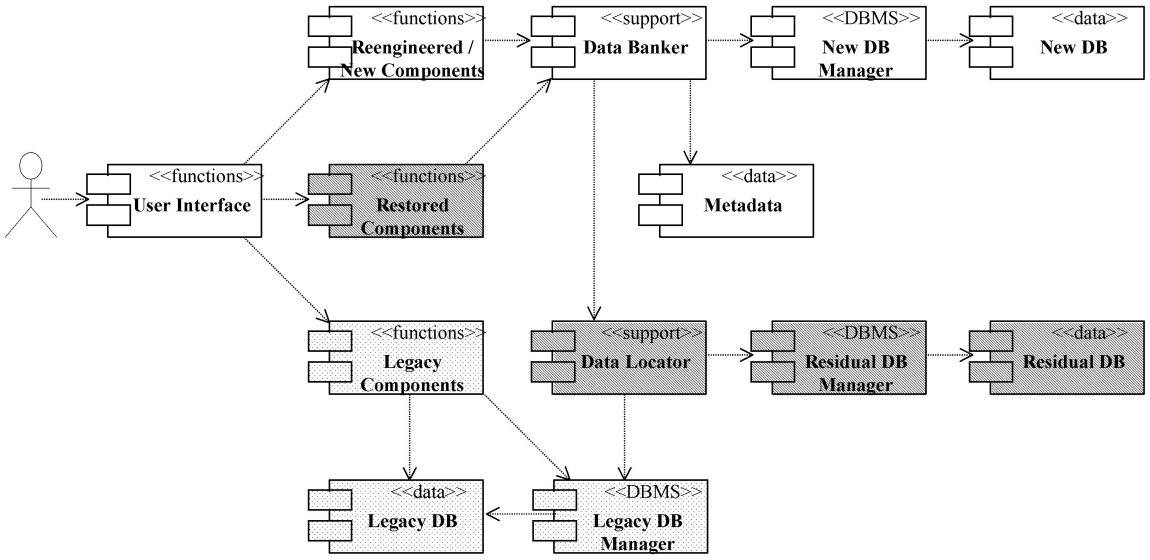

Fig. 2. The system architecture during reengineering.

through the database management system used by the aged system. The access of Legacy Components to data in Legacy DB is managed by a "LEGACY DB MANAGER." Note that, in general, in legacy systems, such a manager could be embedded in the Legacy Components, and it may not be a conceptually well-defined component. Nevertheless, we show it as a separate component for the sake of clarity.

The component labeled "NEW DB" indicates the database with the new structure. It will include all primary data here migrated from the Legacy DB and all data, which are primary for the functions added to the system during reengineering. It exploits the features afforded by modern database technologies, that will enable more efficient management through an updated database management system, shown in Fig. 2 as the set of components labeled "NEW DB MANAGER." The residual data of the Legacy DB will be stored in the "RESIDUAL DB." It is managed by the "RESIDUAL DATA MANAGER," that may be either the new database management system or that of the legacy software system. When the "REENGINEERED COMPONENTS" and the "RESTORED COMPONENTS" need data, they require them from the "DATA BANKER." The Data Banker, through "METADATA," will know if the required data are stored in the New DB, in the Residual DB, or in the Legacy DB. In the first case, it will retrieve their physical location always through Metadata. The knowledge of the physical data location will then be used by the Data Banker to require data off the New DB Manager. In the other two cases, Data Banker will route the request to the "DATA LOCATOR." The Data Locator, after interpreting the information the Data Banker has retrieved from Metadata, will require the appropriate data respectively off the Residual DB Manager or the Legacy DB Manager.

It should be noted that the architectural components in Fig. 2 are shown with three different degrees of shading, representing three different life spans. The components with light shading ("LEGACY COMPONENTS," "LEGACY DB MANAGER," and "LEGACY DB") are the original components destined to gradually disappear as the process goes on; those with darker shading are temporary components that allow the procedures to be reengineered after the data; the components with no shading are those that will remain at the end of the process, and that will make up the reengineered system. It is worth noting that both the DATABANKER and METADATA packages will remain after reengineering. METADATA knows the New DB structure, while the Data Banker knows the services provided by the New DB Manager and the ways to access them. This means that, if the physical structure of the NEW DB changes, only data in METADATA need to be changed, while if the New DB Manager changes, only the Data Banker needs to be changed.

3.3 Process

The reengineering process presented in this section is based on previous experiences by the same authors [22], [23]. The process activity diagram is shown in Fig. 3: Note that, in it, only the main paths are indicated. In the following, each phase of the process will be further detailed.

3.3.1 Analyze Legacy System

During reengineering of a legacy system component, all the requests for change that have an impact on this component must be put on hold until the component has been reengineered. For this reason, the system is partitioned into components so that the reciprocal interdependencies,[1] i.e., the client-supplier relationships between components, are minimized. This allows both the number of change requests which have to be frozen and the freezing times to be minimized. Therefore, this partitioning has the aim of identifying the components in the legacy system that minimize the impact of the reengineering. So, the time that change requests are put on hold is kept to a minimum.

1. According to UML terminology [26], a dependency indicates a client/supplier relationship: The client depends on the supplier to provide certain services. While this relationship holds, the client operations invoke supplier operations.

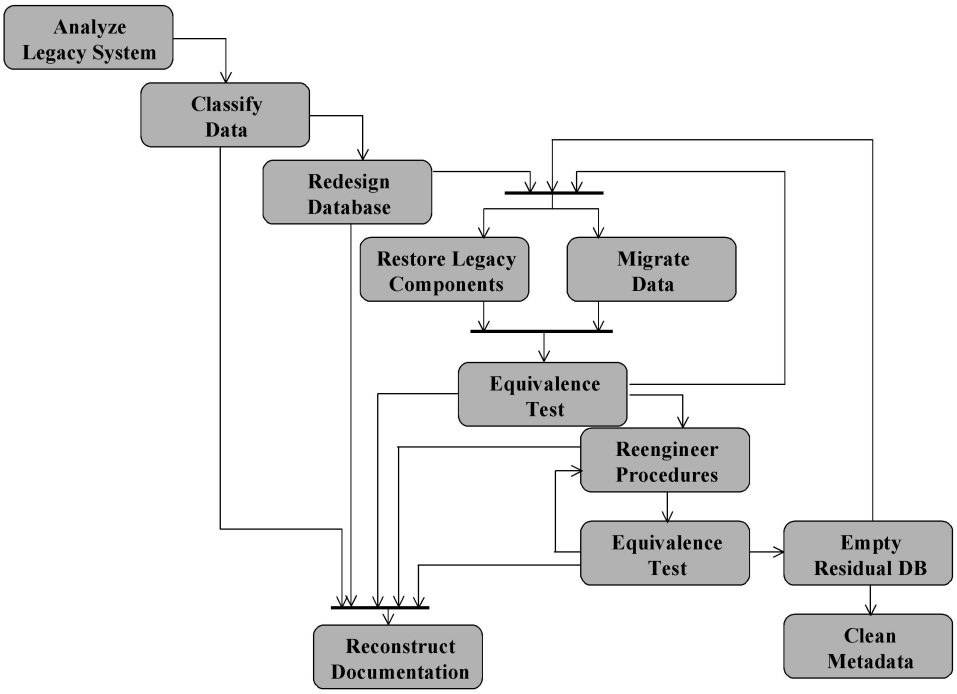

Fig. 3. The Iterative Reengineering process.

If n_i is the total number of requests for maintenance made for the ith component and if $\Delta t_{j,i} = t_{j,i} - t_{j-1,i}$ is the time interval between two successive requests for maintenance of the ith component, then, in order to partition the components to be reengineered, we introduce the *mean time to maintenance request* for the ith component ($\mathsf{MTMR_i}$) according to the following formula:

$$MTMR_i = \sum_{j=1}^{n_i} \frac{\Delta t_{j,i}}{n_i}.$$

The values for n_i and $\Delta t_{j,i}$ are obtained from the historical archives of the system. In many cases, data about n_i and $\Delta t_{j,i}$ may not be available because they were not recorded systematically, for example. In all these cases, such data must be established by making use of the experience of the project team: Each member tries to assess the data on the basis of their respective knowledge of the history of the legacy system, and the project manager produces a synthesis of all the assessments, from which $\mathsf{MTMR_i}$ can be derived. When the expected time for reengineering the ith component ($\mathsf{RT_i}$) is less than $\mathsf{MTMR_i}$, then it is reasonable to suppose that maintenance requests for this component are unlikely to be received during the time it is being reengineered. The greater the $\mathsf{RT_i/MTMR_i}$ ratio, the higher the probability that it will be necessary to freeze maintenance requests for the ith component. Therefore, if $\mathsf{RT_i}$, is higher than $\mathsf{MTMR_i}$, then the component must be divided into subcomponents, each having a better $\mathsf{RT_i/MTMR_i}$ ratio. It is worth noting that the use of MTMR is only one of the possibilities, which help when deciding the priorities to follow in reengineering. More sophisticated methods could take into account the variance, or the analysis of trend. These topics, as well as partition driven by experience and the related pitfalls, are outside the scope of the present work, for which the information provided by MTMR is enough.

The preliminary phase of analysis of the legacy system also involves identification and analysis of the usage relationships between the data files and the various components in order to establish the best way to reengineer the individual components. For example, in the case study we executed, at first glance the legacy system Fa2000 can be partitioned into three main components: the business functions package, which manages the access to all the conceptual data; the user interface package, which manages data input/output; and the support package, which includes all the programs supporting the system operations. The three components obtained by this breakdown have a high value for the $\mathsf{RT_i/MTMR_i}$ ratio (greater than 16), and are strictly interrelated; therefore, each one of them should be further partitioned into smaller components. The business function package can be subdivided into each of the business functions, so as to obtain a component for managing the pharmacological products store, one for the nonpharmacological products store, a third one for managing relations with National Health Service, and so on. Again, the business functions granularity is too coarse, with respect to both the $\mathsf{RT_i/MTMR_i}$ ratio values and the reciprocal interdependencies, therefore each component is further decomposed into programs. The function managing the relationship with suppliers is executed by a number of programs: For the sake of simplicity, we will only consider the programs named FATMPB.CBL, FDITTB.CBL, and FTABEB.CBL. Each of these programs presents a good $\mathsf{RT_i/MTMR_i}$ ratio, not more than 1.8. Therefore, this granularity satisfies the constraint to minimize the freezing time for maintenance requests.

TABLE 1
An Example of Cross Reference Programs-Data Files Accesses for Components to be Reengineered

	FATMPB.CBL	FDITTB.CBL	FTABEB.CBL	SUBSSN.CBL	FPFARB.CBL	FPNUMB.CBL
ARCCOD	R			R	CRUD	
ARCFED	R		CRUD	R		
DAT080	R	R	CRUD			
DAT240	R		CRUD			
DATFAR	R	R	R	R	CRUD	
DATFED	R	R	CRUD			
DATFOR	R	CRUD				
PARSTA		R			CRUD	
DATPRO						CRUD
DATPOS						CRUD
DATCAT				R		CRUD

Note that reengineering the programs FATMPB.CBL, FDITTB.CBL, and FTABEB.CBL also has an effect upon the files they access, i.e., the data files ARCCOD, ARCFED, DAT080, DAT240, DATFAR, DATFED, DATFOR, and PARSTA. But, there are three other programs accessing these data files: SUBSSN.CBL, FPFARB.CBL, and FPNUMB.CBL. Therefore, in order to satisfy the reciprocal interdependencies constraint, all these programs have to be considered together because of their common files. Table 1 summarizes the files accessed for each of the previous programs, and also indicates the access mode: creation (C), reading (R), updating (U), and deletion (D). So, if the set of programs {FATMPB.CBL, FDITTB.CBL, FTABEB.CBL, SUBSSN.CBL, FPFARB.CBL, FPNUMB.CBL} is considered as a single component, then its RT/MTMR ratio is 2.34, which is an acceptable value. In fact, the expected reengineering time for this set of programs is 100 hours, therefore it is expected that no more than three maintenance requests should need to be frozen during its reengineering.

In the case study, this phase was carried out with the support of the MicroFocus Revolve tool 5.0 [27]: The tool was used to establish the usage relationships between the data files and the programs.

3.3.2 *Classify Data*

This phase involves identification and interpretation of the data recorded in the Legacy DB and of their reciprocal relationships. During this phase the data are also classified according to the definitions given in the "Background" section. At the end of the data classification phase, a table is obtained that records all the nonduplicated data present in the Legacy DB. Table 2 is an example of an extract of such a table.

In the table, "datfor-dare" and "datfor-avere" are primary conceptual data which respectively define the provider's debts and credits: This information is needed for executing some business functions. Indeed, "datfor-saldo" is a residual datum, and more precisely a computationally redundant datum, in that it expresses the balance and is calculated as the difference between the value of "datfor-dare" and "datfor-avere," "datfor-saldo" will be stored in the Residual DB until all legacy programs using it have been reengineered. "datfor-arrot-far" and "datfor-arrot-par" are used, respectively, to indicate if the amount of a pharmacological or nonpharmacological product ordered is greater than a threshold: They are examples of control data in that they indicate whether a given discount can be applied. "datfor-pdc" is an example

TABLE 2
An Excerpt of Data Classification in the Legacy DB

Data Name	Data Type	Description
datfor-dare	Primary - Conceptual	Amount due to provider
datfor-avere	Primary - Conceptual	Amount owing from provider
datfor-anno-autorizza	Primary - Conceptual	Year when legal authorization was obtained
datfor-anno-revoca	Primary - Conceptual	Year when legal authorization was revoked
datfor-arrot-far	Residual - Control	Indicates if the amount of a pharmacological product ordered is greater than a threshold
datfor-arrot-par	Residual - Control	Analogous to the above, but with respect to non-pharmacological products
datfor-codice-banca	Primary Conceptual	Code identifying the bank
datfor-pdc	Residual Semantically redundant	Bank code number
datfor-sconto	Primary - Structural	Primary key used to access discount data
datfor-sco	Residual Structurally redundant	Discount code used to identify discount ranges
datfor-base	Residual - Control	Flag indicating the official Register the pharmacological product is recorded in
datfor-key	Primary - Structural	Primary key used to access provider
datfor-saldo	Residual Computationally redundant	Balance with respect to provider

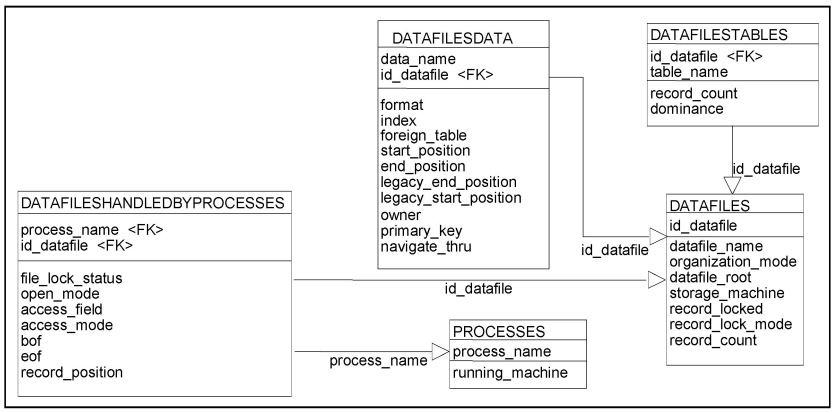

Fig. 4. An example of Metadata tables.

of a residual semantically redundant datum, in that it stores the same information as the primary conceptual datum "datfor-codice-banca." Finally, "datfor-sconto" and "datfor-sco" are two examples of structural data, used to organize data structures in the legacy system. The former becomes a primary datum in the New DB, as it is the primary key used to access discount data. The latter is a redundant structural datum and it will be stored in the Residual DB until the completion of reengineering of all the programs using it, then it will be removed.

In order to classify data, they need to be correctly understood. Therefore, this phase allows to update or create the documentation concerning data dictionary. In particular, attention must be paid to the documentation of primary data because it will be inherited by the reengineered system.

This phase was also carried out in the experiment with the support of MicroFocus Revolve 5.0 [27].

3.3.3 Redesign Database

During the Redesign Database phase the data classified as primary in the previous phase must be restructured so as to adapt their structure to the new database management system and to remove all the defects of the legacy database, i.e. control and redundant data. For example, in Fa2000, the Legacy DB is organized in a hierarchical approach: During the phase, the New DB is designed adopting a relational approach [28] through normalized tables on the basis of the first five normalization levels ([29], [30], [31], for example). We used Smith's normalization technique [31].

During this phase, while defining the access mode to the new database, the software engineer also defines the characteristics of all data which have to be stored in the Metadata database. Finally, this phase allows any existing redundancies to be observed and redundant data to be eliminated.

Fig. 4 shows an example of five tables in Metadata: DATAFILESDATA indicates, for all legacy data, the reference to a field in the New or Residual DB; DATAFILESTABLES includes all the tables in the New DB corresponding to data files managed by Metadata; DATAFILES includes all the tables in the New DB corresponding to data files managed by Metadata; DATAFILEHANDLEDBYPROCESSES expresses the cross reference between data files and active programs during system execution; *PROCESSES* includes all the programs accessing the tables in the New DB through Metadata.

This phase produces the documentation concerning the design of the New DB. Its execution requires human knowledge and abilities, which cannot be formalized; therefore, it is not supported by any tool, except for the use of a graphical editor to draw the dependency diagram among data.

3.3.4 Restore Legacy Components

Before reengineering a component, it is necessary to redirect the access of all data dealt with by the component itself; moreover, it is necessary to execute this activity for all components dealing with the same data as the previous one. For example, referring to Table 1, when FATMPB.CBL has to be reengineered, all data included in the files it accesses (i.e., all data included in ARCCOD, ARCFED, DAT080, DAT240, DATFAR, DATFED, and DATFOR) should be migrated into the New DB and Residual DB, according to the results of the Redesign Database phase, and they should also be registered in Metadata. In order to pursue this result without altering the operations of the software system, it is necessary to restore all the programs accessing the same data as FATMPB.CBL, if not previously restored, i.e., FDITTB.CBL (which accesses DAT080, DATFAR, DATFED, DATFOR), and FTABEB.CBL (which accesses ARCFED, DAT080, DAT240, DATFAR, and DATFED), and SUBSSN.CBL (which accesses ARCCOD, ARCFED, and DATFAR).

The Restore Legacy Components phase aims to make the legacy system programs compatible with the reengineered data. For this purpose, within each legacy system program that accesses the data to be reengineered, all the instructions involved in accessing the data must be identified. They must then be replaced by new ones that, instead of accessing the data directly, call on the data banker for this service: This will be the component accessing data on behalf of the calling program.

Fig. 5 shows an example of adaptation of a legacy component. In Fig. 5a, there is the piece of code belonging to the legacy program and, in Fig. 5b, the corresponding piece of code after adaptation. More precisely, the piece of legacy

Legacy Program

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

(a)

Adapted Program

```
.
.
LETTURA-DATFOR-DIRETTA.
    CALL "LEGGI" 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 SALDO.
    GO DITTA-SELEZIONATA.
:
```

(b)

Fig. 5. An excerpt of code (a) before and (b) after adaptation.

code includes a label ("LETTURA-DATFOR-DIRETTA." i.e., 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 SALDO (i.e., balance). Finally, it jumps 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 service request to the Data Banker, which is implemented by a call to the new program named LEGGI (i.e., "read"). This program requires the parameters ID-DATFOR (the identifier of the firm), DATFOR (the file to be read), and LEGACY-FILE-STATUS (a return parameter indicating the current status of the legacy file) to be passed. 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 SALDO. Finally, there is a jump to the label "DITTA-SELEZIONATA." Note that neither the code "00" or "99" can be changed during adaptation, as they are COBOL codes expressing the success or failure, respectively.

After the adaptation, the system can act in one of two ways depending on the users' request:

1. Execute the legacy system procedures that access the Legacy DB if the request does not involve reengineered data.
2. Execute the procedures with the restored components that operate on the reengineered data.

Thanks to the Data Banker, this dual possible behavior of the system is transparent to users, who will continue to operate as they did formerly with the legacy system. The correct choice is executed by the User Interface component in Fig. 2. During experimentation of the method, the instructions for accessing the data were identified with the support of the MicroFocus Revolve 5.0 tool [27], while the programs were updated using the development environment Acucobol, version 4.3 [32].

3.3.5 Migrate Data

As the procedures are adapted, in order to keep them operative, it is necessary to migrate the data they use to the New DB or Residual DB, on the basis of the actions executed during the Redesign Database phase. For example, Fig. 6 shows an extract of the datafile DATFOR of Fa2000 Legacy DB, in which data are organized as sequential records. DATFOR is the archive that contains data related to the pharmacy suppliers. It contains a primary key (datfor-key) made up of three data datfor-codice, datfor-divisione, and datfor-base; an identifier (datfor-sigla); a description (datfor-descriz); and so on. Both primary (for example, datfor-dare and datfor-avere) and residual data (datfor-saldo) appear. Data related to this file have partially migrated to the New DB organized in tables, as shown in Fig. 7 and are partially in the Residual DB, also structured in tables, as shown in Fig. 8. For example, data in the Legacy DB concerning the address of the supplier (i.e., datfor-tipo-indir, datfor-indir, datfor-cap, datfor-citta,

```
fd  datfor.
01  datfor-rec.
    03  datfor-key.
        05  datfor-codice            pic xxxx.
        05  datfor-divisione         pic xx.
        05  datfor-base              pic x.
    03  datfor-sigla                 pic xxx.
    03  datfor-descriz.
        05  datfor-descr-1           pic x(18).
        05  datfor-descr-2           pic x(12).
    03  datfor-fiscale               pic x(16).
    03  datfor-email                 pic x(50).

    03  datfor-stato                 pic xxxx.
    03  filler                       pic x(8).
    03  datfor-pdc                   pic 9(8).
    03  datfor-max.
        05  datfor-fasce-max         pic s9(5) occurs 5.

    03  datfor-tipo-indir            pic x(25).
    03  datfor-indir                 pic x(30).
    03  datfor-cap                   pic x(5).
    03  datfor-citta                 pic x(25).
    03  datfor-prov                  pic xx.
    03  datfor-telefono              pic x(20).
    03  datfor-fax                   pic x(20).

    03  datfor-partita               pic 9(11).
    03  datfor-pagamento             pic 999.
    03  datfor-banca                 pic 999.
    03  datfor-dare                  pic s9(11).
    03  datfor-avere                 pic s9(11).
    03  datfor-saldo                 pic s9(11).
```

Fig. 6. An example of data included in the Legacy DB.

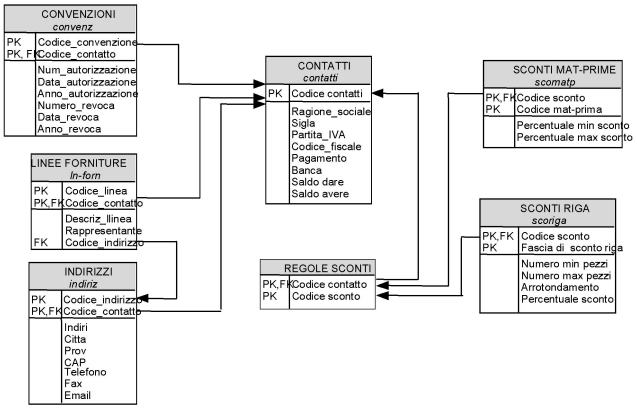

Fig. 7. An example of an excerpt of the New DB.

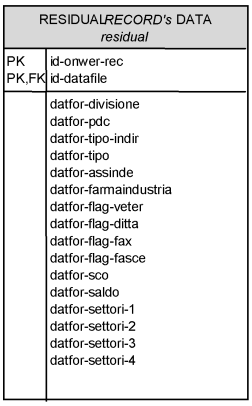

Fig. 8. An example of an excerpt of the Residual DB.

datfor-prov, datfor-telefono, and datfor-fax) migrated to the New DB table named INDIRIZZI, in which some fields are unchanged (indir, cap, citta, prov, telefono, fax), some are not present, having migrated to the Residual DB (datfor-tipo-indir), and some have been added (Email).

Note that, to make it clear that the data no longer depend on a specific file, their names have been modified in the New DB: The significant part is maintained but the prefix that specified the file it belonged to in the Legacy DB has been removed. On the other hand, data migrated to the Residual DB have preserved their original names (datfor-tipo-indir).

In the experiment, a Data Migrator tool developed ad hoc in the Software Engineering Research LABoratory (SER_Lab) of Bari University was used. For each data file being reengineered, this tool reads all the data it contains and, according to the information contained in the Metadata, copies them into the Residual DB or the New DB.

3.3.6 Reengineer Procedures

In the Reengineering Procedures phase, the functions are reengineered, causing them to evolve from a *restored* to a *reengineered* state. During this phase, the software engineer analyzes the quality deficiencies of procedures and introduces suitable remedies. More precisely, the software engineer:

- restructures the components to bring their quality up to the desired standards; particular care has to be paid to information hiding and to all features that make maintenance easier;
- individuates any procedures of the component being reengineered that are clones of already reengineered procedures, and eliminates such clones in favor of the best quality procedure;
- updates data access management to match the new organization;
- improves the algorithms used;
- updates the modules interface;
- updates the user interface;
- executes the maintenance operations that had been put on hold while reengineering;
- updates the programming language to more modern versions.

Each of the above operations must be carried out reusing the components originally present in the legacy system as much as possible. This strategy is prompted not only by economical reasons but also in the interests of preserving the skills the maintainers have developed while operating on the system, in accordance with the fifth Lehman's Law on software evolution [3]. It is assumed that the maintenance team that operated on the legacy system is also likely to operate on the reengineered system and that it is therefore desirable to preserve such familiarity with the system as is compatible with the updating process. It is worth noting that this phase also produces the documentation concerning the system design. For the sake of clarity, in the following, examples of the effects of reengineering of a procedure on management of data access, on updating of the user interface, and on execution of a maintenance request are reported.

Data access management. While a procedure is being reengineered, the decisions about how to represent information in the data or procedures made for the legacy system can be revised, in order to eliminate redundant computational data as far as possible. For example, the instruction MOVE X4 TO SALDO in the code excerpt in Fig. 5b has been replaced by instructions which take into account the New DB structure; therefore, all the values for SALDO have been cancelled from the Residual DB and the SALDO field has also been eliminated from Metadata (Fig. 9).

Updating the user interface. Fig. 10 shows a screen display of the legacy version of Fa2000. User interaction with systems having this kind of interface is difficult [33] and can give rise to various kinds of errors. Reengineering should also update the interface and system interaction modes. In the reengineered Fa2000 system, the user interface was updated, being changed to the Windows approach (Fig. 11).

Note that, in the legacy version, both the character-based user interface and the commands supplied by the functions key had to be managed by the programmer; instead, in the reengineered version, they are managed by the programming environment used for the reengineering process.

Maintenance execution. The highly dynamic nature of the application domains of aged legacy systems leads to a great number of maintenance requests. During reengineering, the system manager must therefore expect to devote a large part of the effort to maintenance operations.

```
.
.
LETTURA-DATFOR-DIRETTA.
    CALL "LEGGI" ID-DATFOR DATFOR LEGACY-FILE-STATUS, DARE, AVERE.
    EVALUATE LEGACY-FILE-STATUS
        WHEN 00 GO PRENDI-DITTA-ALFABETICA.
        WHEN 99 GO LETTURA-DATFOR-DIRETTA.
    END-EVALUATE
    SUBTRACT AVERE FROM DARE GIVING SALDO.
    GO DITTA-SELEZIONATA.
:
```

Fig. 9. An excerpt of code in a reengineered program.

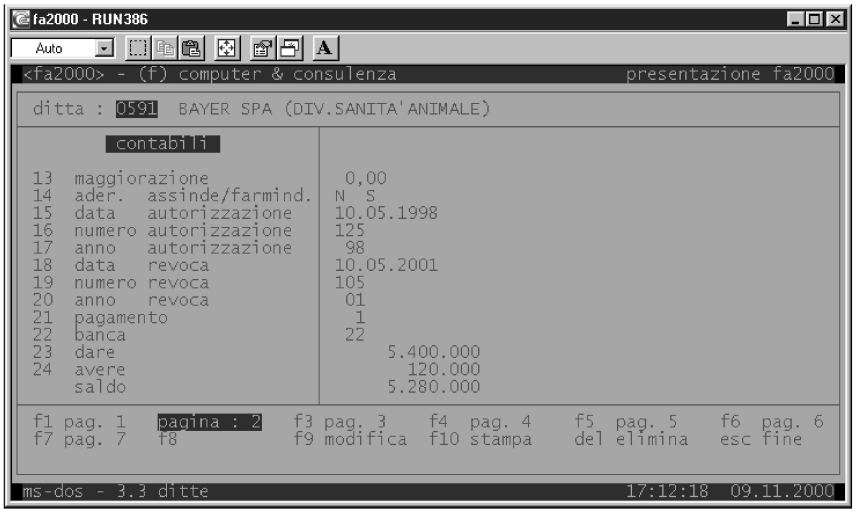

Fig. 10. A display screen of the original management of a supplier's data.

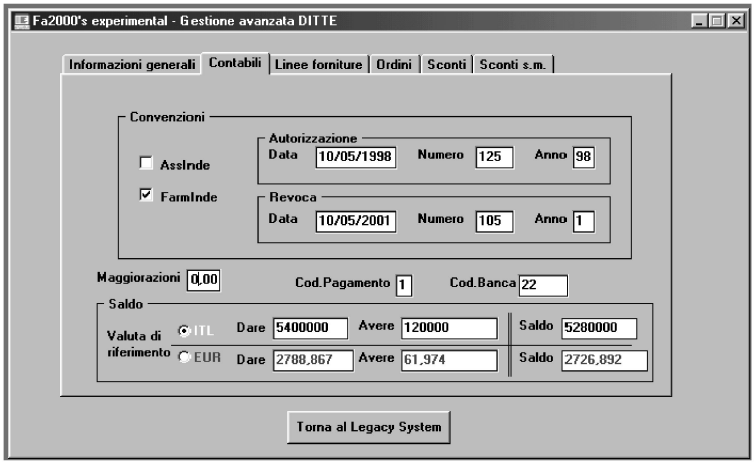

Fig. 11. A display screen of the reengineered management of a supplier's data.

In our case study, one of the maintenance operations required on the system was currency management. In fact, in the legacy system, the amounts are expressed in Italian liras, as shown in Fig. 10. It was therefore necessary to update the system to express the dual currency regime becoming the norm in Italy in the year 2002. The maintenance operation carried out enabled the amounts to be expressed in the dual currency, Italian lira and euro, as shown in Fig. 11.

3.3.7 Equivalence Tests

The equivalence test [34] aims to ensure that the behavior of the software system after a maintenance activity of one or more of its components is exactly the same as before the change. This guarantees that after rejuvenation the software system can continue to correctly execute all its operations. Therefore, this test is necessary after data migration and procedures restoration, and after procedures reengineering. Moreover, this phase allows the documentation concerning the test plan to be rejuvenated. This test is executed taking into account a sample of transactions considered to be important by the users; it is chosen during the system execution-on-field. The expected results are the same results as the system produced before the maintenance. The execution of the test is iterated until the system produces the expected results for all the transactions selected.

3.3.8 Empty Residual DB

The iterative reengineering allows migration of each reengineered part of the Legacy DB: some data into the New DB, some other into the Residual DB. Data included in the Residual DB are no longer necessary in the overall system when all the procedures of the Legacy Components needing them have been reengineered. Therefore, in this case, data which are no longer useful should be removed from the Residual DB. For example, referring to Table 1, residual data extracted from DAT080 when reengineering FATMPB.CBL should remain alive until completion of the reengineering of FTABEB.CBL. The Residual DB will be *completely* empty by the end of the reengineering process, as no functions will access its data. For this reason, the Residual Data Manager and Data Locator components can also be eliminated from the system, as they will have become superfluous.

3.3.9 Iteration

Once a given component of the legacy system has been reengineered, the process is repeated, applying it to the next component, until the whole legacy system has been reengineered.

3.3.10 Clean Metadata

When the whole legacy system has been reengineered, Metadata should include only data concerning data recorded into the New DB during the Redesign Database phase. All data concerning Residual DB data have become useless; therefore, they must definitively be removed. Moreover, all the Data Banker procedures, which define the access to the Data Locator, and the access itself, should be removed as all the data managed by the reengineered system are included in the New DB. For example, in the excerpt of Metadata shown in Fig. 4, the fields concerning the legacy start and end position are removed during the Clean Metadata phase.

3.3.11 Reconstruct Documentation

The Reconstruct Documentation phase has been indicated only for the sake of completeness, as it is outside the scope of this work. This phase makes explicit the fact that the system reengineering proceeds together with the whole documentation reengineering. In this sense, Fig. 2 emphasizes the reconstruction of documentation after executing Classify Data, Redesign Database, Equivalence Test, and Reengineer Procedures phases with the aim of preserving traceability to the current system state. Nevertheless, it is worth stressing that the need of up-to-date documentation can be satisfied all along the reengineering process [35]. If the documentation concerning the system requirements is out-of-date or incomplete, then reverse engineering of the documentation produced by the reengineering process is needed.

4 Case Study

The described iterative reengineering process was applied to the system Fa2000. This is an important benchmark for experimenting the method since it is a legacy system that has been in use for a long time and that requires improvements of its maintainability. In fact, the system has undergone a great number of maintenance operations during its life span and they have contributed to degrade its quality. Because of the changes carried out, the system features many unstable components due to dynamic evolution of the application domain. The legacy system features are measured by metrics, which include, if possible, the metrics used by developers and maintainers. This aims to make the meaning of measures more comprehensible for developers and maintainers.

4.1 Fa2000 Characterization

Fa2000 is a support system for pharmacies management, distributed in approximately 100 pharmacies in Italy. The data it manages refer to the chemical companies producing the products the pharmacies deal with, pharmaceutical chemistry aspects of the products, and the health, economical, and legal issues associated to them. The system development started in 1987 and the first version, for the UNIX platform, was distributed since 1989. Its application domain is subject to specific, highly dynamic regulations and, as a consequence, many frequent maintenance requests were made, approximately one request per week. The same company that developed the first version carried out the corrective, adaptive, or perfective changes executed over the years; these maintenance operations included migration of the system to the MS-DOS environment in 1991. The version used for the experimentation was released in December 1999, running on the MS-DOS platform.

From a quantitative characterization viewpoint, the Fa2000 version used for the experimentation consists of 2,312 modules, expressed as the sum of Cobol paragraphs, sections, and external routines, for a total of 600 KLOC. A total number of 350 data files are managed by the system, among which 8,000 record types are stored for a total of 970,000 fields.

As to the system's complexity, the *Integration Cyclomatic Complexity* (*ICC*) of the whole system and the *Mean Cyclomatic Complexity* (*MCC*) per module are considered. If C_i is the cyclomatic complexity of the *i*th module and M is the total number of modules, then ([36]):

- $C_i = e_i - n_i + 2$, where e_i is the number of links in the flow graph of the *i*th module (i.e., computational statements or expressions in the module), and n_i is the number of nodes (i.e., transfer of control between nodes).
- $MCC = \frac{\sum_{i=1}^{M} C_i}{M}$.
- $ICC = \left(\sum_{i=1}^{M} C_i\right) - M + 1$.

In the legacy version of Fa2000, ICC=2,540, while MCC = 2,098. These low values for cyclomatic complexity are due to the specific features of the legacy system. In fact, Fa2000 is a strongly data-oriented system, the functions mainly deal with data management, and the computation algorithms are few and they require few decision points. Therefore, from the point of view of cyclomatic complexity, the legacy system does not feature specific problems.

4.2 Aging symptoms of Fa2000

The system shows a number of aging symptoms, which have been aggravated by the continuous maintenance

TABLE 3
Summary of the Classification of Data in the Legacy System Fa2000, Before and After the Reengineering

Files	Legacy System		Reengineered System	
	Absolute Value	Percentage	Absolute Value	Percentage
Temporary	35	10	0	0
Pathological	280	80	0	0
Problem free	35	10	287	100
TOTAL	350	100	287	100

TABLE 4
Summary of the Classification of Data in the Legacy System Fa2000, Before and After the Reengineering

Class of Data	Legacy System		Reengineered System	
	Absolute Value	Percentage	Absolute Value	Percentage
Semantically redundant	97,000	10	0	0.00
Control	87,300	9	15,300	1.99
Structurally redundant	77,600	8	0	0.00
Computationally redundant	116,400	12	0	0.00
Conceptual and Needed Structural	591,700	61	753,700	98.01
TOTAL	970,000	100	769,000	100.00

activities, as stated in [3]. Besides the adaptive maintenance interventions caused by the changes in the application domain throughout its years of operation, Fa2000 has also been subjected to many corrective maintenance interventions, which have determined a general decay of the entire system's quality.

The reengineering process phases: Analyze Legacy System and Classify Data make it possible to identify the values the metrics assume for the legacy system and to detail the system's aging symptoms, as shown in [2]. As to the values of the Fa2000 files, there is a significant number of temporary files (35 files, equal to 10 percent of the total) and pathological files (280 files, equal to 80 percent of the total), while only the remaining 35 (10 percent of the total) are problem free.

In general, the high presence of temporary files is due to the need to establish communication between two or more subsystems within the same system, when this communication could not be achieved with the original database. The management of all these files makes the data management procedure harder and, therefore, makes system maintenance more burdensome [2].

The presence of pathological files is due to bad system design and development, carried out without appropriately applying software engineering principles [2]. The presence of pathological files also underlines the aging symptom of coupling among system components. This aging symptom, in turn, makes impact analysis of the change very difficult during maintenance. Moreover, this symptom causes difficulties in designing the system tests due to the difficulty in identifying the best path to test and in defining the state of the database most appropriate for test execution.

Considering the data more closely, Fa2000 includes 378,300 fields dealing with residual data (equal to 39 percent of the total) and 591,700 fields dealing with primary data (61 percent). Application of the redundant data classification previously described yields 97,000 fields for semantically redundant data (10 percent of the total data), 87,300 for control data (9 percent), 77,600 for structural redundant data (8 percent), and 116,400 for computationally redundant data (12 percent). The high number of residual data confirms the presence of the aging symptoms layered architectures and coupling, in Fa2000. In fact, the structurally, semantically, and computationally redundant data point out that it suffers from the layered architectures symptom, while the control data highlight the coupling aging symptom.

4.3 Fa2000 Rejuvenation

The reengineering process led to a reduction of the aging symptoms featuring coupling and layered architectures, so confirming the experimental results presented in [2]. First, note (Table 3) that reengineering removed from Fa2000 all the temporary and pathological files. This result was obtained thanks to the "Redesign Database" and "Redesign Functions" of the process adopted. These phases also led to a reduction of the total number of files, thanks to both removing a number of unused data and better organization of the New DB.

Table 4 shows a classification of the data before and after executing the reengineering process. The reengineering process made it possible to eliminate all the fields concerning semantically, computationally, and structurally redundant data and, thus, eliminate the layered architectures symptom from the reengineered system. It was not possible to eliminate all the control data because in some cases they were required to keep track of the asynchronous events related to the system and make a decision on the basis of these events. For example, in the case of datfor-arrot-far and datfor-arrot-par described in Table 2, the number of (pharmacological and nonpharmacological) products ordered is registered: The pharmacy is entitled to a discount only if this exceeds a certain limit.

However, the reengineering process resulted in a drastic decrease in the number of fields concerning control data in terms of both percentage (from 9 percent to 1.99 percent of the total) and absolute values (from 87,300 to 15,300). The presence of control data persisting in the system after reengineering is a negligible detail. Therefore, the coupling symptom is also absent in the reengineered version of

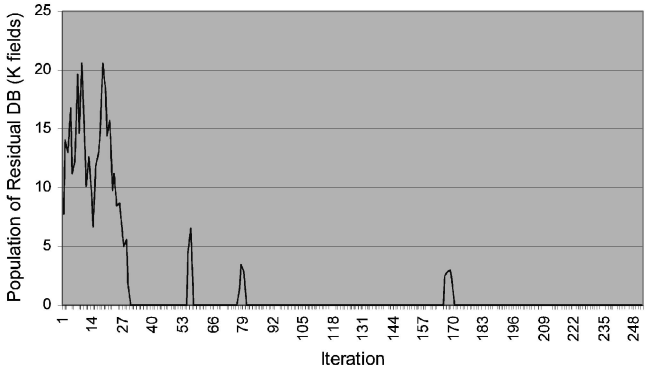

Fig. 12. Progress of the Residual DB population during the reengineering process.

Fa2000. It is worth noting that the New Database also has a lower total number of fields (769,000 versus 970,000 fields in Legacy DB), although the number of primary data has increased in comparison to the legacy database because some fields have been added to realize the temporal coordinates required to replace the removed temporary files. Moreover, 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 primary data are also used as primary keys. The reduction of the set of data managed by Fa2000 was obtained by continual filling and emptying of the data included in the Residual DB, due to the combined effect of the Redesign Data and Empty Residual DB phases. If Ins_i is the number of fields inserted at the ith iteration, Rem_i is the number of fields removed from the Residual DB at the ith iteration, Pop_i is the number of fields which populates the Residual DB at the ith iteration, then the following holds:

$$\mathrm{Pop_i} = (\mathrm{Pop_{i-1}} + \mathrm{Ins_i}) - \mathrm{Rem_i}.$$

Fig. 12 shows the Residual DB population fluctuations during the reengineering of Fa2000 at each iteration i. More precisely, it is indicated the number of fields in the Residual DB at each iteration. It should be borne in mind that not all the data contained in the Residual DB are necessarily removed after each iteration, so the residual population is inherited by the successive iteration: Only when the process ends does the Residual DB become empty. The figure shows that most of the fields were transferred to the Residual DB during the first iterations. In particular, from a total of 378,300 fields transferred to this DB, 349,100 were inserted and deleted within the first 30 iterations, while the following 29,200 were transferred within the 55th and the 170th iteration. The remaining 80 iterations did not require data migration to the Residual DB.

Reengineering improved the system's complexity: Table 5 summarizes the values for the metrics: Number of Modules, LOC, Cyclomatic Complexity, and Mean Cyclomatic Complexity of Fa2000 calculated with MicroFocus Revolve 5.0 before and after reengineering.

Of the reduction in the number of modules from and the corresponding reduction in lines of code:

- 40 percent was due to different management of the interface, that was carried out by specific procedures in the legacy system; whereas, in the reengineered system, it is carried out by components belonging to the new middleware.
- 35 percent was due to the elimination of clones.
- 25 percent was due to the reduction of data managed.

As to the cyclomatic complexity, although the legacy version already had favorable values, reengineering further improved the values of the two metrics as a side effect of reducing the control data, decreasing them from 87,300 to 15,300. This allowed for the removal of the selection instructions conditioned by these data. It should also be noted that the percentage reduction in the value of integration cyclomatic complexity is higher than the reduction in mean cyclomatic complexity, in that ICC is affected by the selection instructions regulating coordination among the modules [36]. Reengineering greatly simplified this coordination, thanks to the reduction in the total number of modules, so that the ICC value improved much more than the MCC value.

The reengineered Fa2000 system was tested on 1,928 test cases. Of these, 1,062 cases were designed to test data management after executing the Migrate Data phase and 866 cases were designed to test the functions after executing the Redesign Functions phase. In total, 356 test cases were positive: 250 for restoring and 106 for reengineering.

The effort required for reengineering was 6,904 person/hours and it was spent in 250 iterations of the process described in the previous sections for a total calendar time of 18 months. Table 6 summarizes all data related to the effort, expressed in person/hours arranged by phase.

Note that the "Analyze Legacy System" phase had a relatively low cost, due to the fact that it was executed with the project administrator's support. He had an overall view of the legacy system and this made comprehension of the system by the software engineers easier. The "Classify Data" phase was particularly costly because the software engineers executing the case study had first to understand

TABLE 5
Values of the Complexity Metrics for Fa2000 Before and After Reengineering

Metric	Legacy System	Reengineered System
Number of Modules	2,312.000	705.000
LOC	600,000.000	100,000.000
ICC	2,540.000	512.000
MCC	2.098	1.725

TABLE 6
Summary of the Effort Spent on Each Fa2000 Reengineering Phase Expressed in Person /Hours

Phase	Effort (person/hours)	Percentage
Analyze Legacy System	400	6
Classify Data	680	10
Redesign Database	1,160	17
Adapt Legacy Components	720	10
Migrate Data	200	3
Equivalence Test (restoring)	600	9
Reworking (restoring)	640	9
Redesign Functions	1,624	23
Equivalence Test (reengineering)	592	9
Reworking (reengineering)	288	4
TOTAL	6,904	100

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 help given by the project administrator in this phase had less influence because an understanding of details concerning code was requested. The phases which required most effort were "Redesign Database" and "Redesign Functions." In the former case, a further comprehension of the programs with reference to the dependencies among data was first required and, then, it was possible to design the database on the basis of the architecture provided. In the latter case, it was not only necessary to reengineer the functions, but also to execute the required maintenance activities. The effort required for executing the "Equivalence Test" phase, for both restoring and reengineering, was affected by two main factors: The legacy system test cases selected were among the most frequent transactions used by the organization. Due to the iterative nature of the process, it was necessary to operate on small components, which required a small number of equivalence tests, each time.

Note that in Table 6 the effort spent on reconstructing the documentation is not explicitly indicated, in that this activity is spread throughout the whole process and it is not possible to isolate it. Finally, note that during the reengineering period, which took a total calendar time of 18 months, 98 maintenance interventions were requested. Of these, 63 were frozen for less than 10 working days, 28 for between 11 and 15 days, and only 7 for between 15 and 28 working days.

5 Conclusions

This work presents an iterative model for reengineering aged legacy systems. The proposed architecture allows coexistence between the new and old legacy systems during the process. In this way, the users can continue operating with the *system* as a whole, accessing both the reengineered and the legacy databases. To this end, only 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, stored in a single new database, besides their own new data.

The iterative approach adopted offers the typical advantages of the *divide et impera* techniques: The problem is divided into smaller problems, which are easier to manage. In the specific case, the system's dimensions being reengineered are reduced at each iteration.

Moreover, the iterative process reduces the freezing time for the maintenance requests that emerge during the process execution. In fact, if these requests do not refer to the components being reengineered, the maintenance can be done immediately, while reengineering other components. On the other hand, if they refer to components involved in reengineering, the requests are frozen for less than the time necessary to conclude the cycle. In each case, this time is less than the time that would be required if the process were to involve the entire system in a single cycle.

The coexistence between new and old legacy systems during the process and, therefore, the possibility of using the system while reengineering proceeds, allows the proposed method to be applied throughout the life span of the system and not only when it needs to be rejuvenated. In this sense, the iterative reengineering process can be considered as a process for *continuous improvement* of the quality of the software system. Thanks to this feature, the method can be integrated into some other methodologies, such as the *Refactoring* technique, for example. This consists of changing a software system in order to improve its internal structure without altering its external behavior [37]. Its main weaknesses have to do with improving the system data structures, but the solution to this problem is the core of the iterative reengineering method, as described in this paper.

Experimentation in an industrial environment with the aim of evaluating the overall effectiveness of the proposed method was successfully carried out. The data obtained demonstrate that the legacy system was kept working throughout the reengineering process. Moreover, only few requests for change had to be held up and this delay lasted only few days. Finally, an example of an adaptive change realized during reengineering is described.

The experience on field shows the dynamics of the residual data and those transferred to the new database, demonstrating that the legacy and the new database could coexist. There are also some examples showing the coexistence of two operative systems and three systems of functional components, the legacy, the restored, and the reengineered systems. Moreover, the proposed method can be applied to software systems written in whatever programming language and running in whatever environment. Only the tools supporting the reengineering process

depend strictly on the programming language and the operative environment of the legacy system.

It should be noted that in our case study we did not take into account performance loss [21] due to the introduction of the data banker. Besides this, the main weaknesses of the approach are:

- The need to build and maintain the data locator, which is removed at the end of the process. This weakness can be 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.

Finally, it is worth noting that the authors are proceeding with their research lines concerning software reengineering. In fact, in [2], a quality model for software aging symptoms is presented. This work shows that the reengineering process presented is able to solve the following aging symptoms: pollution, coupling, and layered architectures. In fact, the values of all the metrics detailing these symptoms improved after execution of the reengineering process. It cannot be claimed that the process solves all aging symptoms, but we can surely state that the process rejuvenates the system.

The size of the case study presented is relevant: 2,312 modules, 600 KLOC, 350 data files, 8,000 record types, and 970,000 fields. Nevertheless, the cause-effect relationship between execution of the reengineering process and improvement of the symptoms cannot be definitely validated because a model for quantifying the ability of iterative reengineering to rejuvenate aged systems is not available. Therefore, more on field experimentation will be necessary. The authors are involved in further experimentation aiming to provide a model which quantifies the metrics improvement and are willing to collaborate with other researchers wishing to replicate the experiment.

ACKNOWLEDGMENTS

The authors would like to thank all the students who participated in the case study for their fruitful work and, above all, T. Baldassarre for her patience. We are also very thankful for the diligent and efficacious contribution made by R. Kudlicka, administrator and maintainers' manager of the legacy system, which greatly aided understanding of the application. Special thanks go to Mary V. Pragnell for her contribution as a technical writer. Finally, we are grateful to the anonymous reviewers for their interesting suggestions, comments, and remarks.

REFERENCES

[1] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy Information Systems: Issues and Directions," *IEEE Software,* vol. 16, no. 5, pp. 103-111, Sept./Oct. 1999.

[2] G. Visaggio, "Ageing of a Data Intensive Legacy System: Symptoms and Remedies," *J. Software Maintenance and Evolution,* vol. 13, no. 5, pp. 281-308, 2001.

[3] M.M. Lehman and L.A. Belady, *Program Evolution–Processes of Software Change.* London: Academic Press, 1985.

[4] W.B. Noffsinger, R. Niedbalski, M. Blanks, and N. Emmart, "Legacy Object Modeling Speeds Software Integration," *Comm. the ACM,* vol. 41, no. 12, pp. 80-89, Dec. 1998.

[5] P. Robertson, "Integrating Legacy Systems with Modern Corporate Applications," *Comm. the ACM,* vol. 40, no. 5, pp. 39-46, May 1997.

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

[7] A.J. Brown, "Specification and Reverse Engineering," *Software Maintenance Research and Practice,* vol. 5, pp 147-153, 1993.

[8] M.R. Blaha, "On Reverse Engineering of Vendor Databases," *Proc. IEEE Fifth Working Conf. Reverse Eng.,* pp. 183-190, 1998.

[9] M.R. Blaha, "An Industrial Example of Database Reverse Engineering," *Proc. IEEE Sixth Working Conf. Reverse Eng.,* pp. 196-203, 1999.

[10] H.M. Sneed, "Planning the Reengineering of Legacy Systems," *IEEE Software,* pp. 24-34, Jan. 1995.

[11] H.M. Sneed, "Encapsulating Legacy Software for Use in Client/Server System," *Proc. IEEE Third Working Conf. On Reverse Eng.,* pp. 104-119, 1996.

[12] F.P. Coyle, "Does COBOL Exist?" *IEEE Software,* vol. 17, no. 2, pp. 22-36, Mar./Apr. 2000.

[13] F.P. Coyle, "Legacy Integration Changing Perspectives," *IEEE Software,* vol. 17, no. 2, pp. 37-41, Mar./Apr. 2000.

[14] A. Quilici, "Reverse Engineering of Legacy Systems: A Path Toward Success," *Proc. 17th Int'l Conf. Software Eng.,* pp. 333-336, Apr. 1995.

[15] E.R. Hughes, R.S. Hyland, S.D. Litvintchouk, A.S. Rosenthal, A.L. Schafer, and S.L. Surer, "A Methodology for Migration of Legacy Applications to Distributed Object Management," *Proc. Int'l Enterprise Distributed Object Computing Conf.,* pp. 236-244, 1997.

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

[17] H.M. Sneed, "Encapsulation of Legacy Software: A Technique for Reusing Legacy Software Components," *Annals Software Eng.,* vol. 9, pp. 293-313, 2000.

[18] S. Comella-Dorda, R.C. Seacord, K. Wallnau, and J. Robert, "A Survey of Black-Box Modernization Approaches for Information Systems," *Proc. Int'l Conf. Software Maintenance,* pp. 173-183, Oct. 2000.

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

[20] B. Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O'Sullivan, "The Butterfly Methodology: A Gateway-Free Approach for Migrating Legacy Information System," *Proc. Int'l Conf. Eng. Complex Computer Systems,* pp. 200-205, 1997.

[21] H.M. Sneed, "Risks Involved in Reengineering Projects," *Proc. IEEE Sixth Working Conf. Reverse Eng.,* pp. 204-211, 1999.

[22] A. Bianchi, D. Caivano, and G. Visaggio, "Method and Process for Iterative Reengineering Data in a Legacy System," *Proc. Seventh IEEE Working Conf. Reverse Eng.,* pp. 86-96, 2000.

[23] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative Reengineering of Legacy Functions," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 632-641, 2001.

[24] M.R. Blaha and W.J. Premerlani, "Observed Idiosyncracies of Relational Database Designs," *Proc. IEEE Second Working Conf. Reverse Eng.,* pp. 116-125, 1995.

[25] UML Revision Task Force, "OMG Unified Modeling Language Specification, version 1.3,"Technical Report, document ad/99-06-08. Object Management Group, June 1999.

[26] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[27] MERANT,"MicroFocus Products—Revolve,"http://www.merant.com/products/microfocus/revolve/,2000.

[28] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. the ACM,* vol. 13, no. 6, pp. 377-387, 1970.

[29] R. Fagin, "Normal Forms and Relational Database Operators," *Proc. ACM–SIGMOD,* pp. 153-160, 1979.

[30] W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," *Comm. the ACM,* vol. 26, no. 2, pp. 120-125, 1983.

[31] H.C. Smith, "Database Design: Composing Fully Normalized Tables from a Rigorous Dependency Diagram," *Comm. the ACM,* vol. 28, no. 8, pp. 826-838, 1985.

[32] AcuCORP, "AcuCOBOL—GT," http://www.acucorp.com/Solutions/acucobol-gt.html, 2000.
[33] J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey, *Human-Computer Interaction.* Addison-Wesley, 1994.
[34] B. Beizer, *Software Testing Techniques.* Van Nostrand Reinhold, 1983.
[35] G. Visaggio, "Value-Based Decision Model for Renewal Processes in Software Maintenance," *Annals of Software Eng.,* vol. 9, pp. 215-233, 2000.
[36] A.H. Watson and T.J. McCabe, "Structured Testing: A Design Methodology Using the Cyclomatic Complexity Metric," *NIST Special Publication 500-235,* D.R. Wallace, ed., NIST Contract 43NANB517266, Sept. 1996.
[37] M. Fowler, *Refactoring—Improving the Design of Existing Code.* Addison Wesley, 1999.

Alessandro Bianchi received a degree in computer science at the University of Milan in 1990. After graduation, he was technical coordinator of the Telematics Laboratory of the Computer Science Department in the same University. Then, he worked for some years in the Pisa Applied Research Laboratory of Data Management S.p.A. (FINSIEL Group). In 2000, he received the PhD degree in information engineering from the University of Brescia. In 1999, he joined the Software Engineering Research Group of Department of Informatics, University of Bari, Italy, where he currently is an assistant professor. His research interests deal with experimental software engineering, mainly related to software maintenance, component-based software engineering and software architectures. He served as a Program Committee member for the IEEE International Conference on Software Maintenance. He is a member of the IEEE Computer Society

Danilo Caivano received a degree with honors in informatics at the University of Bari, Italy. After graduating, he started his PhD degree in software engineering at the Department of Informatics in the University of Bari, working within the Software Engineering Research Laboratory (SER_Lab). His research interests are mainly focused on software process improvement and software maintenance. Currently, he is collaborating on several research projects and carries out controlled and in field experimentation within small and medium enterprises. He is a member of the IEEE Computer Society.

Vittorio Marengo received a degree in business administration at the University of Bari, Italy. He is currently an associate professor at the same university, responsible for masters degrees in "Theory and Techniques of Electrical Commerce over the Internet." He is the head of the Experimental Laboratory of Informatics Applied to Business Games. He collaborates jointly with the Machine Learning Group of University of Bari and with the Universitè Paris-Dauphine (France) —Group de Apprentissage Numerique—Symbolique. He has also been a visiting lecturer at the University of Belgrano–Buenos Aires (Argentina). His research interests are twofold: On one hand, he investigates methodologies for designing Intranet corporation networks, and on the other hand he is studying data mining methods related to inferential statistics.

Giuseppe Visaggio received a degree in electronic physics at the University of Bari, Italy, in 1972. After graduating, he continued to work in the same university in computer science and became a professor at the Informatics Department in the University of Bari. His research interests are in maintenance focusing particularly on processes, quality improvements, and legacy systems. Currently, he is a full professor of software engineering at University of Bari. He is the chief of research at the Software Engineering Research Laboratory (SER_Lab), in the Informatics Department at the University of Bari. SER_Lab hosts several basic research projects and carries out controlled and on the field experimentation. For many years, he has worked as a member of the program committee for IEEE International Conference on Software Maintenance (ICSM), Workshop on Program Comprehension (WPC), and Workshop on Empirical Studies of Software Maintenance (WESS). Since 1998, he has served on the Steering Committee of ICSM. He is member of the IEEE and IEEE Computer Society, ACM, and AICA (the Italian Computer Society).