JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2001; **13**:281–308 (DOI: 10.1002/smr.234)

Research

Ageing of a data-intensive legacy system: symptoms and remedies

Giuseppe Visaggio*,†

Department of Informatics, University of Bari, Italy

SUMMARY

This study generalizes some of the symptoms of ageing of a legacy system. Each symptom is specified by metrics and the results of the measurements made suggest what operations should be undertaken to renew the software. The study is based on retrospective analysis of data collected during the execution of a large renewal process of a very old legacy system. It therefore provides evidence of the expected efficacy of such renewal processes and can be used to decide how best to plan them and manage them in order to increase their efficacy. It can also be used to define the reengineering requirements to ensure long life to the system despite successive evolutions of the application and the operation domain. The metrics can provide a basis for monitoring a software system to ensure that its quality does not degrade to such an extent that the most costly and risky renewal processes then have to be performed to improve it. Finally, the paper points out the problems with renewal processes that still remain open. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: renewal process; legacy system; comprehension; maintenance; reverse engineering

1. INTRODUCTION

A legacy system is generally one of an organization's assets with a high economic value. Even when it ages, it is both difficult and risky to replace it. Difficult, because the system is used by many people within the organization and its replacement would involve retraining all the users to understand the new system. Risky, because the construction or purchase of a new system may go over budget and disrupt planned schedules; moreover, the new system may lack some functions the users of the previous system were used to having.

Even when the legacy system is new and well-engineered, software maintenance tends to cause its quality to degrade; hence system ageing [1,2]. As the software system ages, maintenance becomes more costly and less accurate. In short, the system becomes progressively less maintainable and

*Correspondence to: Professor Giuseppe Visaggio, Department of Informatics, University of Bari, Via E. Orabona, 4-70126-Bari, Italy.
†E-mail: visaggio@di.uniba.it

Copyright © 2001 John Wiley & Sons, Ltd.

Received 13 September 2000
Revised 1 June 2001

this reduces its economic value [3]. If its maintainability diminishes below acceptable limits for the owner organization, the legacy system will have to be replaced by a new one. To avoid this need for replacement, which would be costly and risky, it is necessary to monitor ageing symptoms and treat them while such an operation is still possible with only a relatively small effort.

For this reason, the main aims of the present work are: to define a list of ageing symptoms, individuate the respective observation metrics and indicate the relative improvement operations to be undertaken to treat the symptoms. The experimental evidence is derived from a retrospective analysis of data collected during a renewal process of a very aged legacy system whose execution required great effort.

The experimental evidence also shows the efficacy of the improvement operations that can be executed and therefore indicates which types of operation do not improve the value of the legacy system compatibly with the effort required to carry them out. Symptoms requiring this type of operation must be prevented to avoid irreparable ageing of the software system.

The symptoms indicate the conceptual aspects of the ageing of the software, at a high level of abstraction and do not, therefore, directly indicate the operative aspects, i.e. the antidotes to software ageing. For this reason, each symptom is then operatively specified by product quality metrics. The values of these metrics express the degree of ageing of the system and their interpretation suggests what activities should be included in the renewal process to improve its quality and remedy the ageing symptoms.

This generalization in terms of symptoms and remedies serves to make the experience gained with the case study analysed in this work transferable and reusable. Clearly, when reusing this experience, it must be tailored to the characteristics of the user organization.

The study is addressed to practitioners because it advises them what to measure to monitor ageing symptoms, what operations are necessary to treat the symptoms and what the expected efficacy of the operations is. It is also addressed to researchers because it stimulates further experimentation of the renewal process, to add to the list of symptoms and metrics, and also indicates the weak points of present renewal process models and technologies and thus suggests possible research areas.

In the case study, the findings resulting from the *post mortem* of the renewal process carried out were generalized to derive the symptoms and relative metrics. Instead, this paper first presents the symptoms and metrics and then describes the measurements, to provide the experimental evidence the generalization is based on. This is because the symptoms, metrics and relative improvement operations are more interesting for the reader, as they are reusable.

To ensure a clear understanding of the paper, the meaning ascribed to some key words in this work needs to be defined: the term legacy system is synonymous with working software system, i.e. the code together with the documentation which describes its behaviour and structure. When a legacy system has poor quality it is said to be old or aged. The renewal process involves the examination and alteration of an old legacy system to improve its quality. This process encompasses a combination of subprocesses such as reverse engineering, restructuring, restoring, reengineering, retargeting, migrating and so on. In the present work, the renewal process taken into account comprises reverse engineering and restoring.

Reverse engineering is the process of analysing a subject system to identify its components and their interrelations and to create a representation of the system in another form or at a higher level of abstraction.

Copyright © 2001 John Wiley & Sons, Ltd.

In the restoration process, an abstraction model is restructured to make it easier to read and understand. This restructuring takes account of the semantics of the components of the restored model and is therefore different from that in [4], where it is stated as regards the restructuring process that 'many types of restructuring can be performed with a knowledge of structural form, but without an understanding of meaning'. Instead, restoration in this work means restructuring with an understanding of meaning and knowledge of the structural form. Some aims of this restoration process are: to identify the data which describe the application domain; to extract and eliminate useless instructions; to extract those modules with greater internal cohesion.

The reengineering process is also mentioned in this work, to be distinguished from that of restoration. Reengineering involves redesigning the application and modifying, where necessary, the design choices and thus the relationships between components. Although the need for this process is indicated for the relative symptoms, the process is not described in detail as it was not carried out in the case study.

From now on, the term renewal process will be taken to mean the combination of reverse engineering and restoration. Details and their respective histories of improvement can be found in [5], although these are not necessary to understand the present work. The definitions of the renewal phases are based on [4] but some changes aiming to increase effectiveness have been made.

This paper includes: related work (Section 2) that also details the reasoning underlying the present work; the description of ageing symptoms derived from generalization of the data collected in the case study (Section 3); the metrics serving to identify the ageing symptoms, whose measurements provide the basis for the decisions as to which quality improvements should be made to the legacy system (Section 4); the case study demonstrating the significance of the metrics defined and showing those symptoms that are little affected by the operations that can be carried out in a renewal process (Section 5). Some discussion of renewal processes is included to provide an overview of the characteristics of the process from the points of view of productivity and cost (Section 6). The work concludes with some suggestions for those that manage legacy systems, already aged or still young, and points out some of the problems left open, addressed to researchers (Section 7).

2. RELATED WORK

The scientific and industrial communities have devoted considerable attention to maintenance processes. Apart from the many studies presented in the journals and conferences on software engineering and at international software congresses, there is a specialist journal, and conferences and workshops are held on the subject.

In view of this vast scientific production, the following quotations should be taken as only a few examples. Some studies have gone into the theoretical aspects of comprehension e.g. [6], others have studied the various aspects of the renewal process to improve software maintenance e.g. [4,7–12] and the technologies supporting this process e.g. [13–19].

Instead, to the author's knowledge, only a few studies have studied the symptoms, causes and antidotes to software degradation. Some works have experimented with metrics for classifying the components of a legacy system and individuating the most risky from the point of view of quality degradation e.g. [20]. In these works, the main aim was to identify the most significant metrics from empirical evidence.

Copyright © 2001 John Wiley & Sons, Ltd.

The works most similar to this one are [21–23], in which the authors list various symptoms. However, the symptoms listed are often different, and some are called by different names despite having the same meaning. The reason why some symptoms are different is because the experimental evidence they are based on is different, while the synonyms are due to the fact that the approach is not consolidated and so the symptom names are not universally recognized. Unlike in the present work, in these previous works the symptoms have not been detailed by metrics; moreover, they do not include the experimental evidence from which the symptoms were derived, even if these were based on a real case study.

The symptom-based approach is spreading; in [24], metrics are used to specify symptoms in a different application domain from the present one. In fact, from the [24] experience, the authors aim to extract useful suggestions for new software development processes.

Another new aspect of the present work is the investigation of the causes that generate the symptoms, from which the most efficacious remedies are then derived.

3. AGEING SYMPTOMS

A legacy system ages when its maintainers have difficulty in evolving it, or are quite unable to introduce the evolutions required to satisfy changes in the application domain or to perfect the system.

This section presents a list of symptoms derived from retrospective analysis of a large renewal project carried out on a legacy system that had been running for many years. Each symptom is detailed by

- its conceptual description,
- the causes generating it and
- its effects on the maintenance process.

3.1. Pollution

Many components of the software system (programs, reports, files or data) present in the legacy system do not serve to carry out the application functions exploited by the users.

This symptom is caused by poor quality of the software configuration process. During the system's life span, some functions were created that remained valid for a limited period of time, after which the components that carried out these functions failed to be eliminated. Moreover, during maintenance of the system, solutions eliminating the logical parts of the data and code were adopted but these were not physically cancelled.

Pollution complicates analysis of the impact of a change because it extends the potential impact by including useless software parts. The most important side-effect is the increased complexity of the specifications for the change and hence the greater risk of inaccuracy when detailing these. Moreover, in those particular cases where the software components do not correspond with the business functions requiring them, it is impossible to carry out changes or evolutions of these functions.

3.2. Embedded knowledge

The knowledge of the application domain and its evolution is spread over the programs and can no longer be derived from the documentation. This knowledge can be partly reconstructed by reading the programs with the aid of the maintainers and program users.

Copyright © 2001 John Wiley & Sons, Ltd.

Very often, after changing the programs, maintainers do not update the documentation adequately. The software system then becomes inconsistent because the documentation does not record the real structure of the code, nor the decisions the maintainer made when carrying out the changes.

The lack of traceability of the software system makes maintenance more troublesome owing to the difficulty of defining the impact of each change, and hence increases the risk of making inaccurate changes. Often, since the maintainer is unaware of the existence of one of the legacy system's functions, instead of modifying this one he/she creates a new one to conform it to the evolution of the application domain. This increases the redundant code and causes a new disease: system ageing due to pollution.

3.3. Poor lexicon

This is present when the names of variables and components have little lexical meaning or are in any case inconsistent with the meaning of the components they identify.

This symptom is caused by the use of idiomatic practices of doubtful utility. For example: the name must not be longer than n characters; the name of a variable must be preceded by a mnemonic identifier of the structure it belongs to and so on. These practices limit the name of a variable in such a way that it cannot explain the meaning of the component. Moreover, over the years this practice tends not to be homogeneous owing to the different interpretations and applications made by the various different maintainers dealing with the same software system. This heterogeneity increases the lexical inconsistencies of the software.

Software maintenance is more difficult as a result of the reading and comprehension difficulties generated by this symptom.

3.4. Coupling

The programs and their components are linked by an extensive network of data or control flows. This symptom is generated when insufficient effort is spent on controlling the entropy of the software that inevitably results from the maintenance process.

High coupling makes any change very costly. Often, despite great effort, the maintainer is unable to make a precise analysis of its impact, even when using automatic tools. In addition, the complexity of the code makes it difficult to carry out the regression test after each change because it is hard to individuate which paths should be tested and to define the most adequate state of the database for the execution of the tests. Thus, the modified software will be unreliable.

3.5. Layered architectures

The system's architecture consists of several different solutions that can no longer be distinguished. Even when the software started out with a high quality basic architecture, the superimposition of these other hacked solutions during maintenance damages the quality.

The symptom is due to the software engineer's need to deal with variations in the application domain that were not envisaged in the software planning stage. When this occurs, the maintainer takes planning decisions that are then superimposed on the original ones, to satisfy the evolution of the requisites. The result is a software system that becomes ever more difficult to understand during maintenance, especially as regards the reasons for the solutions adopted and where these are localized. This symptom

Copyright © 2001 John Wiley & Sons, Ltd.

also increases the risk of reinserting solutions that already exist in the software. For example: one or more redundant data are updated by different functions at different times and so homologous data will contain different values in the same database.

4. METRICS

The symptoms described above help to understand the problems of ageing of a system but not the operative solutions. In other words, their description does not indicate how they can be observed and treated. It is therefore necessary to detail them by means of metrics that can be measured on the products to be maintained. The value for each metric suggests what operations should be carried out to treat the ageing symptom.

Like the symptoms described above, the list of metrics is also based on retrospective analysis of the data collected during the renewal process that was the object of the case study described in this work.

Each metric features the following.

- Its description.
- Its measurement methods.
- The possible remedies. The data analysed were collected during the processes of reverse engineering and restoration; the remedies requiring reengineering are declared, but were not included in the experiment.
- The perfective changes that may be advisable, and which should be carried out during the reengineering process.

The rest of this chapter deals with the metrics featured in this work; no further description of those used that are well known and have a universally-accepted definition in the software-engineers community, such as the cyclomatic number, has been made, to save space.

4.1. Pollution

Duplicate programs. Two programs are considered to be duplicated if they have sources that use the same copies, call the same subroutines, handle the same files in the same way but only one of them has the corresponding executable. The number of duplicate programs can be measured thanks to the use of utilities that compare the content of members of a library. Only the maintainer concerned can identify the most updated version. All duplicate programs must be eliminated.

Obsolete programs. These are programs which have a source but no corresponding executable. These can also be discovered by means of utilities that compare the source libraries with the executable libraries. To link the source to its executable, the procedures used to compile it are analysed, together with the source links. Obsolete sources must also be cancelled.

Sourceless programs. These are executable programs that have no corresponding sources. These programs can be individuated by means of the execution of procedures that generate the executables. In fact, some procedures will require source programs that do not exist in the corresponding libraries. It would be impossible to modify one of these programs owing to their lack of sources. For this reason, in the reengineering process, these programs require rewriting of the source code. The functions each

Copyright © 2001 John Wiley & Sons, Ltd.

must carry out can be ascertained both from their users and from the behaviour of the executable function when it is subjected to test cases to check its acceptability.

Useless components. Many reports that are not used can be produced by the legacy system; these are indicated as *useless reports*. A component is useless if it generates useless reports or creates and modifies files which are used to generate useless reports.

Searching for these components requires a list of paper reports and screen reports; this list is shown to the users, who will indicate the reports they really use. All those that are not identified by any user are useless reports. The components that generate useless reports can be identified by cross-reference.

Useless components sometimes coincide with entire programs that are indicated as *useless programs*, whereas other useless reports are produced by programs which also have other purposes. Thus, for some programs, only the program components dealing with these useless reports must be cancelled; these can be extracted using the *direct slicing* proposed by the author [25]. The components extracted may be modules or sets of instructions scattered among many modules. In either case, cancellation of these components is deferred until the restoration phase, because the code must be restructured accordingly.

To avoid the construction of new components destined to become rapidly useless, it is best to insert a set of functions for *flexible reporting* in the application. These functions must manage the metadata which serve to specify the reports required by the user. When one of the reports defined by the metadata is required, the flexible reports functions adapt their behaviour to the parameters defining the report to be produced. If the user wants a new report, he/she must specify this by recording the relative parameters in the metadata. Instead, when the report becomes obsolete, the corresponding parameters must be cancelled from the metadata. So the flexible reporting functions enable evolutions in the reports produced to be made without changing the programs. The flexible reporting functions may have some performance problems but these can be solved; no description of this operation is provided as it is outside the scope of this paper.

Useless reports constitute an inventory of information which, historically, was required by users and then became obsolete. This phenomenon is due to the variability of the application domain. The information is useful as a basis for analysis, which can also help to forecast future evolutions and to define perfective maintenance to the legacy system to make the software more adaptable to evolutions in the application domain. For this reason, perfective changes that this metric may suggest include the flexible reports functions and structural changes to the programs to improve the software system's adaptability.

Dead data. These are variables created by at least one component of the legacy system but not used by any of its components. They can be individuated automatically using the static analysers that extract the life of each datum in a program. To eliminate them, all the components that create them must be cancelled. These are generally sets of instructions. Cancelling these components is one of the tasks carried out during the restoration process, to identify all the relationships between the instructions to be cancelled and those that remain in the existing programs.

Dead code. These are all those instructions that cannot be reached by the program control flow. They can be individuated automatically, using static code analysers. They must be eliminated, which is easy to do without risking any side-effects because the fact that they are unreachable means they are never executed during the running of the programs that contain them.

Copyright © 2001 John Wiley & Sons, Ltd.

4.2. Embedded knowledge

Incomprehensible data and modules. Variables or modules whose meaning cannot be evinced from the available documentation. The measurement of this metric is made by identifying the data and modules whose meaning is not detailed in the existing documentation or is not consistent with the content of the relative data or modules. This measurement therefore requires reading activities.

The meaning of the data and modules can be extracted from the program documentation, from the users' knowledge of the data and, lastly, from reading the programs to understand how they have been created and used by the applications. Extraction of this knowledge from the human resources and the software documentation transforms the unstable knowledge possessed by the users into stable knowledge that everyone can use. This activity can be carried out during reverse engineering, since no structural change in the programs is required.

Missing capacities. These are application functions that the legacy system contains, but that cannot be precisely localized in the software components. The capacities in the software system are derivable from user knowledge of the programs, from analysis of the reports and files, from the computational data and from reading the documentation the organization employs to regulate correct usage of the applications. For these reasons, this metric also requires reading techniques.

Two slicing techniques can be used to extract the components that create these capacities. If they are functions that serve to collect and control external data, or to extract data from the database and return it in output to the user, then it is best to use direct slicing [25,26]. If, on the other hand, the function is for data transformation, normally a business function, then it is best to use *transform slicing* [12]. This extraction must be done during restoration because it will be necessary to manage the relationships these instructions have with the others in the same program. Once extracted, these slices can be transformed into components that can be automatically recalled and can be included in the new documentation.

4.3. Poor lexicon

Inconsistent data and module names. These are data and modules whose names do not express their meaning. The number of data and modules with inconsistent names can be obtained from the documentation after elimination of the incomprehensible data and modules. This measurement is carried out with reading techniques.

It is necessary to give each datum and module with an inconsistent name a new, more meaningful name and to change the names in the code to the new ones, with the relative utilities. The names for data and modules can be made more meaningful during reverse engineering, since no structural change in the programs is required.

4.4. Coupling

Pathological files. Files created or modified by different programs are called pathological. These files can be identified by means of the programs-files cross reference utility. Unfortunately, to eliminate pathological files, both the procedures and the data in the legacy system need to be restructured, an activity which must be carried out during the reengineering process.

Copyright © 2001 John Wiley & Sons, Ltd.

Control data. These are data that create communications among components to control the behaviour of the components the communication is addressed to. For example, if during running of a program an event occurs that affects the behaviour of another program, then the former warns the latter of the event by recording a control datum in a file they have in common.

These data control the behaviour of the procedures and are therefore used at the decision points about procedures. They can be searched for by identifying all the variables used to express the conditions at each decision point, by means of a static analyser. Each variable in the list that has no connection with the application domain is a control variable. This search can be carried out using the data documentation, and naturally requires reading techniques.

Improvement of this metric can be achieved during the restoration process. The latter, however, can act only on the control variables within the same program. Instead, variables communicated among different programs require reengineering of the database and programs. Moreover, if a program has high complexity, even reducing the control variables within the program will require great effort and the results may not be accurate. In this case it is necessary to reengineer the program. Hence, complete elimination of the control variables can only be achieved with the reengineering process.

Module complexity. In this paper, the complexity of the module is defined according to its number of IF (explicit or implicit in the cycles).

Analysis of the code with a static analyser yields the number and position of the IF statements. Once this list of IF has been extracted, two types should be distinguished.

- *Algorithmic IF* required by the function implemented by the program; the two slices dominated by IF are not single application functions, but both together with IF make up an application function, e.g. 7330 in Figure 1 dominates two slices, neither of which can be given a meaning, whereas the two slices together with IF are the business procedure for dealing with homonyms.
- *Procedural IF* which serve to control the behaviour of the program that includes part of the different functions. If two slices dominated by the same IF are extracted, each will have a complete application function, e.g. the IF of 5760 in Figure 1 dominates two slices, each of which is a validation procedure of the customer's personal data according to type. It is supported by the one in 5850 that individuates two further slices in one of those dominated by the preceding IF. The 5850 also serves to distinguish two procedures for two types of customer that have one part in common, program paragraph labeled 'BB'.

This analysis can be carried out by extracting the slices individuated by each IF with a static analyser and checking that they are complete with reading techniques.

Reduction of the procedural IF is performed in the restoration process, which must manage all the side-effects that their cancellation will generate in the program.

4.5. Layered architectures

Useless Files. A file is useless if no program, or else a useless program, uses it. Useless files can be individuated using the programs files cross reference, ascertaining how the files are used by each program and relating them to the useless programs.

Copyright © 2001 John Wiley & Sons, Ltd.

```
.............
.............
005740 AA.
005750   MOVE SPACE TO NAME-SEC.
005760   IF TT19 NOT = 'JR0000'
005770                    GO BB.
005780
005790   MOVE ZERO TO REL-CODE.
005800   PERFORM READ-ARKAG-SB.
005810   IF NOT KI-0
005820                MOVE '8104' TO FIGS-MESS
005830                PERFORM P-E-M GO EX.
005840   MOVE BUF-NAG TO TT281.
005850   IF TT19 = 'JR0000'
005860                MOVE 'AR0000' TO TT19 GO BB.
.............
.............
007250   ***********************************************
007260   ************* ASSIGNMENT - HOMONYMS *******
007270   ***********************************************
007280 ASSIGNMENT HOMONYMS SECTION
007290 R-O.
007300       MOVE WS-EL-NAG-HOMONYMS (N) TO NAG.
007310       MOVE ZERO TO COD-REPORT
007320       PERFORM READ-ARKAG-SB.
007330       IF NOT KI-0
007340                MOVE 'ASSIGNMENT HOMONYMS' TO
                         NOME-SEC GO EP.
007350       MOVE NAG TO AA0090-NAG (N).
007360       MOVE LEG-NAME TO AA0090 - INT (N).
007370       R-O-EX.     EXIT.
.............
.............
055690       MOVE HONOR-TITLE TO OUT-AG0071-7
055700       MOVE SURNAME TO OUT-AG0071-5
055710       MOVE NAME TO OUT-AG0071-6
055720       MOVE 'AA6700' TO TT19
055730       PERFORM GET-DATE-HOUR-ETC.
...........
...........
```

Figure 1. Program with partially changed names for variables.

Copyright © 2001 John Wiley & Sons, Ltd.

In the same way as with useless reports, the programs that serve only to create useless files must be cancelled; for programs that have other functions it is necessary to extract the parts which serve to build useless files, and cancel them. As above, the first requirement may be satisfied by reverse engineering, because even cancellation of entire programs has no effect on the structure of the legacy system. The second requires restoration because it is necessary to identify all the relationships among the parts of the program that have been cancelled and the rest of the program.

Such files may have been used for memorizing temporary data required to build information used over a brief time span. At the end of this time, the programs built to read these data and construct the information the user needs were cancelled from the libraries, whereas the files they used remained in the database, while the functions that generated and modified them sometimes also remained active. Analysis of the useless files can help to understand which evolutions of the application domain could not easily be assimilated by the database structure, and so to forecast the perfective changes best suited to the database structure.

Obsolete files. A file is obsolete if the software uses it but has no programs for creating new records in it, except in the possible situation where the file contains information that is exogenous to the system and is therefore imported by means of utilities that are not included among the software system's functions.

These files can be individuated using the program-files cross reference utility, ascertaining how the files are used by each program and relating them to the context of the legacy software, in order to be able to inventory those files that are exogenous to it.

It is necessary to cancel the programs which serve only to read, modify or cancel records in obsolete files and to extract and cancel modules or sets of instructions with these purposes from programs having others. Again, program cancellation can be performed during the reverse engineering process while the other activities belong to the restoration process.

The existence of these files may be due to cancellation of the programs which created them because they implemented functions no longer required by the application. In this case, too, analysis of the files can point out some functions that have become obsolete as a result of evolution of the application domain and so help to forecast the most variable areas in the application domain. Perfective changes in the architecture of the legacy system may be required to make the most volatile functions easily modifiable.

Temporary files. A temporary file is created, read but not updated, and deleted by the system. These files can be individuated using the program-files cross reference and ascertaining how the files are used by each program.

In the reverse engineering and restoration processes, this metric cannot be improved. However, the analysis serves to identify requirements for perfective maintenance. These files normally serve to establish communication between two subsystems, when this could not be achieved with the original database. The data in a temporary file are often extracted or calculated from existing data in other files in the same system. In this case study, two situations were observed.

(a) A program Pr_i generates a datum d_{ij} which is live in the legacy system for a period of time decided by its designer. After this decision has been taken, the application evolves, so that the life of the datum needs to be longer than that originally decided by the designer. To solve the problem, a File F_i is created by the maintainer and d_{ij} from Pr_j is stored in this file before d_{ij}

Copyright © 2001 John Wiley & Sons, Ltd.

is destroyed by the program that manages its life span. F_i is often updated to cover new values for the fields it contains; thus for temporary files, the legacy system does not have all the usual file management functions.

(b) A program Pr_i creates data in F_k which will serve for the other programs $Pr_1, \ldots, Pr_k$ that 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 $Pr_1, \ldots, Pr_k$ to give them access to F_i. In this case, too, the file is updated to cover the values of the new fields and so is not fully managed by the legacy system.

Analysis of type (a) files shows that some data in the database must have a nondeterministic life decided by their users, compatibly with the resources available for running the application. Cancellation of a datum could also be decided, following the occurrence of some event. In this case, it is best that the event also be recorded and when it occurred, while the previous value should not be forgotten. It is therefore necessary for such a datum to have a temporal coordinate, and to construct functions which will handle the data with temporal coordinates and identify the events requiring their destruction.

To solve case (b), it is necessary to hide information about the structure of the files in few modules, so that access to new components in the database can easily be inserted as the application domain evolves.

Permanent files. Permanent files are created, used, modified but never cancelled. They can be individuated using the programs files cross reference and ascertaining how the files are used by each program.

This metric cannot be improved with the reverse engineering and restoration processes. However, analysis of the causes for creating these types of files can suggest some perfective changes.

The existence of such files in an old system may mean that the program or programs which dealt with their cancellation have been erroneously modified, eliminating the cancellation instructions; sometimes the content of the file is entirely recreated at each update. To solve these problems, functions for cancelling the records contained in permanent files can be set up. Moreover, as regards files that are entirely recreated at each update, it is necessary to establish whether this is required by the application domain. If, instead, the need is due to a poor design decision, then updating functions must be inserted and those for complete recreation eliminated. Some of these data may have a nondeterministic life and should be treated as described above.

Anomalous files. The records of anomalous files are not created by the application but are read, modified and cancelled. These files, too, must be individuated through the files–programs cross reference utility and the usage method of the files by each program must be ascertained.

The existence of such files shows that all the parts of the application that had the function of creating new records were erased, whereas the other parts of the system that used, cancelled or updated the records in the files were left in existence. The nonexistence of the application parts serving to add new records to these files shows that their content is obsolete, although the applications are able to satisfy the users, so these files have become useless. After having confirmed that they are useless, they are treated in the same way as useless files.

Copyright © 2001 John Wiley & Sons, Ltd.

On the other hand, for files found to be useful, it is necessary to analyse why the application handles them badly, and to redefine the correct requisites for their handling. This problem must be solved in the reengineering process.

Semantic redundant data. Two data are described as redundant as regards the semantic domain if the definition domain of one is contained in, or equal to, that of the other, and if each equal value in the two definition domains can be interpreted in the same way for both data. They can be identified from the updated documentation of the data and from maintainers' and users' knowledge of them.

Elimination of the synonyms involves their substitution with a single datum. This can be done in the reverse engineering phase because it does not require program restructuring. Often, however, synonyms have different attributes (representation, size, type, etc.). In this case elimination of synonyms for different attributes requires a new database design to give the data adequate attributes for all their uses in the application, and to the reconciliation of these with the programs that access them. These are, of course, perfective changes and require reengineering.

Computational redundant data. A datum is described as computationally redundant if a set of data exists $(d_i^1, d_i^2, \ldots, d_i^k)$ in the database where the value of $d_i = f(d_i^1, d_i^2, \ldots, d_i^k)$, f being a calculation function. These data can be found by reading the updated documentation of the data and programs that generate them.

This metric cannot be improved with the reverse engineering and restoration processes. However, analysis of its causes can suggest some perfective changes.

This redundancy clearly reveals a design decision aiming to achieve better performance of the application, that of recording the value of d_i in a file in such a way that it is calculated only when one of the values of the independent data changes, rather than each time the value for d_i is required. Hence, to eliminate this redundancy it is necessary to redesign the database and cancel the data calculated. It is also necessary to insert the functions for calculating them in all the components that use these data. Specification of these last functions can be extracted from the working programs that create the redundant data. In some cases, the instructions that implement them can also be extracted, using slicing techniques.

Structure data. These are data that have no connection with the application domain but support the database structure. They can be recognized by reading the updated documentation of the database.

This metric cannot be improved with the reverse engineering and restoration processes because it is necessary to change the structure of the database: it reveals the need for a perfective change. The database should be reengineered so that its tables conform to the normal form (normalization) so that the structure data remaining in it include only those necessary to unequivocally identify the records in the tables.

The database can become denormalized because it was not designed according to the principles for normal forms. Successive maintenance of the software system, in this case, tends to worsen the structure of the database. Sometimes, instead, the database starts out normalized but the continual changes made introduce normalization defects. On occasion, the denormalization is even voluntarily introduced by the developer on the assumption that this may improve some critical behaviour of the system. This hypothesis may be arbitrary and unproved and could unjustifiably damage the data

Copyright © 2001 John Wiley & Sons, Ltd.

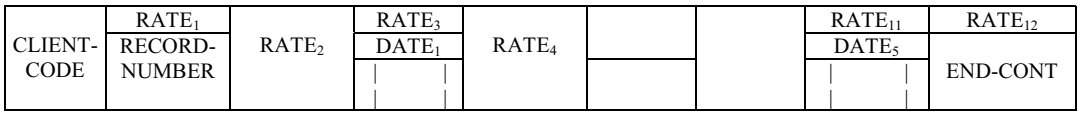

Figure 2. Representation of the two redefinitions of the record for the state of a current account.

base. In any case, when the denormalization is not documented, maintenance will introduce further denormalization defects and the structure data may increase for no valid reason.

The normalization should only take into account data that are meaningful in the application domain (conceptual data). The control and structural data of the old database must not be taken into account because they are architectural supports and may be harmful.

Superimposed data structure. These are different data structures that share the same address spaces. This metric is measured by analysing the inventory of fields that occupy the same addresses in the programs. It cannot be improved with either the reverse engineering or the restoration process but it can suggest perfective changes to the legacy system.

If the superimposed structures are still used by the legacy system then it is necessary to identify the address spaces for each structure and make the consequent changes to the components that manage the database. If some superimposed structures are no longer used, then they must be cancelled and all the necessary changes made to the database and programs that managed them. In addition, analysis of the historical causes that made new structures necessary points out the variable aspects of the application domain that were not adequately assimilated by the architecture of the legacy system. A knowledge of these aspects suggests perfective changes to be made during reengineering of the legacy system architecture.

As this metric is significant in data-oriented systems, an example is provided below. Figure 2 shows the layout of a record whose structure was modified over time, and where some of the structures used had remained superimposed. The first structure included: the CLIENT-CODE and 12 repetitive fields, each of which could contain a rate value; the primary key was the CLIENT-CODE. The second description included: the CLIENT-CODE, the RECORD-NUMBER and five structures with two fields each and a flag; each structure contained a date value and a rate value; the flag value indicated whether it was the last record for this account (END-CONT $=$ 0) or not (END-CONT $=$ 1). The primary key of the new structure was the CLIENT-CODE with the RECORD-NUMBER. All $\{\text{RATE}_n\}$ with $n = \langle 1, 3, 5, 7, 9, 11, 12\rangle$ were used by the new structure to record two data structures (RECORD-NUMBER and END-CODE) and new data (DATE_i) to record the period of validity of RATE_i.

Analysis of this situation showed that, initially, in this domain the variations in rate in one year did not exceed 12. They could be handled by the bank managers, so that each variation in rate would be valid from the beginning of the ith month when this changed. Thus, if the ith field had a value of RATE_i and the jth field of RATE_j, this meant that the account would be calculated as from the beginning of the ith month until the end of the $(j-1)$th month at RATE_i, whereas from the first day of the jth month it would be calculated at RATE_j. Later, rates changed more often and the variation dates were no longer so free, so the record was redefined, associating the starting validity date with

Copyright © 2001 John Wiley & Sons, Ltd.

each new rate value. Since it was no longer possible to forecast the number of rate variations in a year, a list was drawn up with a pointer for the given account. Therefore, the RECORD-NUMBER had to be inserted to complete the primary key and a flag for the last record to indicate the end of the list for this CLIENT-CODE.

The database design needs to be revised to allow this variability of the application domain to be assimilated by the software.

5. A CASE STUDY

The considerations made in this work are of a general nature and are therefore valid in the context of any renewal process. However, the case study carried out is briefly described below, to show the evidence that gave rise to the generalizations that are made in this work. The experimental field needed to be a legacy system of a certain age; such systems are often written in COBOL and are procedural. In any case, it should be borne in mind that despite the specific characteristics of the case, the findings of this study can be generalized to data-intensive legacy systems.

The system under study is a banking application written in COBOL. It is composed of a set of programs, that have a monolithic structure with many internal modules implemented as SECTIONS or PARAGRAPHS and called using PERFORM phrases. Rarely, external modules are accessed with CALL; these external modules are themselves programs in the system. All the data files managed by the system have an index-type organization.

Initially, the system was composed of 16 subsystems, each for a different application domain, making up a library of 6508 programs. The application managed 70 files, all of which were treated, making a total of 9000 fields. There were 1 502 734 procedure division instructions and 12 774 219 data division instructions.

The procedure and data division instructions were counted without taking account of the comments and counting as a single instruction all the clauses of COBOL phrases (e.g. IF . . . THEN . . . ELSE; GO TO . . . DEPENDING ON . . .). One instruction may take up several lines. COPY instructions are considered as a single instruction. The mean age of the programs is 12 years but some are now 23 years old.

The system was so aged that its producer (and distributor) had decided to destroy the existing system and rebuild [27,28]. After one year of rewriting, the risks identified in the process made it apparent that it would be more advisable to try the renewal process.

The measurements corresponding to the metrics described earlier are analysed below, and an interpretation of their values is given, with the aim of transferring the lessons learnt to other contexts with the same problems. All the measurement values are expressed as real numbers and for each, the values obtained after the renewal process are shown.

5.1. Pollution

Analysis of the libraries yielded the results presented in Table I. They show how bad management of the configuration can create pollution and thus make it difficult to recognize the real application in the libraries. Indeed, in this case, the system library was reduced to 639 programs, but of these 14 programs were found to have no existing sources.

Copyright © 2001 John Wiley & Sons, Ltd.

Table I. Measurements of pollution.

Type of program	Cardinality before renewal	Cardinality after renewal
Programs in libraries	6508	639
Duplicate	4323	0
Obsolete	1252	0
Unique sources	933	639
Useless programs	294	0
Sourceless programs	14	14

Table II. Dead and live data in the legacy system.

Type of data	Cardinality before renewal	Cardinality after renewal
Dead data	3597	0
Live data	5403	5403
Total	9000	5403

In addition to these data, 863 useless reports were found. Only 378 of these were eliminated, of which 325 were cancelled together with 294 useless programs. The other 53 reports were produced by programs that had other purposes as well, so their elimination required restoration of the old programs. The remaining 485 useless reports were left because restoration to eliminate them would have been very costly as the programs that produced them were very complex. They needed to be eliminated during the reengineering phase.

The results of the processes carried out on the data are shown in Table II. In addition, 270 507 lines of code (LOC) of dead code were cancelled, on a total of 1 502 734 LOC.

Tables I and II show that the operations to treat the symptom of pollution were efficacious. Two further lessons were learnt as side-effects, which may be useful for those who intend to carry out a renewal process:

Lesson 1. Before renewing an old software system, it is wise to clean it thoroughly of the pollution that has accumulated over the years. The cost of this cleaning will be compensated by the reduction in size of the legacy system, so that the real effort for renewal will be much smaller than the one estimated on the basis of the system before the clean-up.

Lesson 2. The effort spent on restoration may result in a trade-off between the quality targets desired for the renewed programs and the resources available for the restoration process. This trade-off is possible because the renewal process can be carried out gradually.

Copyright © 2001 John Wiley & Sons, Ltd.

The latter lesson was confirmed by all the metrics requiring the renewal process described in this paper. Although this concept will not be repeated again, the reader is asked to recall that each time the restoration process is applied, its depth will depend on the quality targets attained and the expenditure of effort agreed on.

5.2. Embedded knowledge

Before renewal, there were 37 047 modules. In all of them, the existing documentation was insufficient to derive their meaning. Thanks to the organization's documentation and the interviews with users and maintainers, it was possible to assign the relative meaning to most of them; only for 141 was this impossible. The number of business functions listed on the basis of the old documentation and the users' and maintainers' knowledge amounted to 975, for nearly all of which (935), it was not clear which modules implemented them. After restoration, only 25 business functions still could not be traced back to their code.

Improvement operations are not efficacious for this symptom, so it is wise to introduce preventive measures. This is another lesson learnt that may be useful for those planning to execute a renewal process.

Lesson 3. Extracting the knowledge incorporated in the programs is a long and costly activity. It is best to ensure that it is distributed throughout the renewal process and not just concentrated in a single phase. Moreover, it should be extracted before proceeding to carry out those restoration activities which require the knowledge as a preliminary to their execution. For this reason, a continuous documentation updating process should be set up parallel to the renewal process. Despite this, not all the knowledge embedded in the software can always be extracted, as part of it is lost over time as different users and maintainers get to work with the same system.

5.3. Poor lexicon

In the case study it was necessary to change the names of all 9000 data and all the preexisting modules during the renewal process. Of course, names were also given to all the new modules created during the process. Including old and new modules, 36 976 names were given. Just to give one example, Table III shows the data dictionary that traces the new name back to the old one for each datum.

To show the consequences of poor lexicon on the readability of the programs, Table II reports the listing of a program where the old names of the variables have been partly replaced by new names. It is evident that the new names improve the readability.

Lesson 4. It is best to check the lexical quality of the programs quite frequently during the life of the software and, in cases of deterioration, deal with this straight away to avoid side-effects on the maintenance process.

5.4. Coupling

There were 57 pathological files in the legacy system that coupled a high number of programs, as shown in Figure 3. As many as six files were created and modified simultaneously by 40 programs, while these 40 programs were not the same 40 for all six files. Files were most often shared by three programs.

Copyright © 2001 John Wiley & Sons, Ltd.

Table III. Excerpt from data dictionary.

New name	Meaning	Classification	Dimension	Domain	Old name
CODE–ABI	Italian Banking Association code used when client is a credit company	Conceptual entity CORPORATE BODY	Code	Numeric-Code5	AGN-L1 into ARKAG Record type 1
CODE–RISK	Code assigned by risks centre to client having one or more credit limits from any bank on national territory	Conceptual entity CLIENT	Code	Code11	AG22E into ARKAG Record type 1
DATE–UPD–ADD-INSTALMENT	Date for comparison with instalment-due date for debiting relative sum to current account. Date is introduced by operator on activation of a batch. Procedure operates on loans with automatic debiting order	Control			MT00-2 into ARKMT Record type 000

Copyright © 2001 John Wiley & Sons, Ltd.

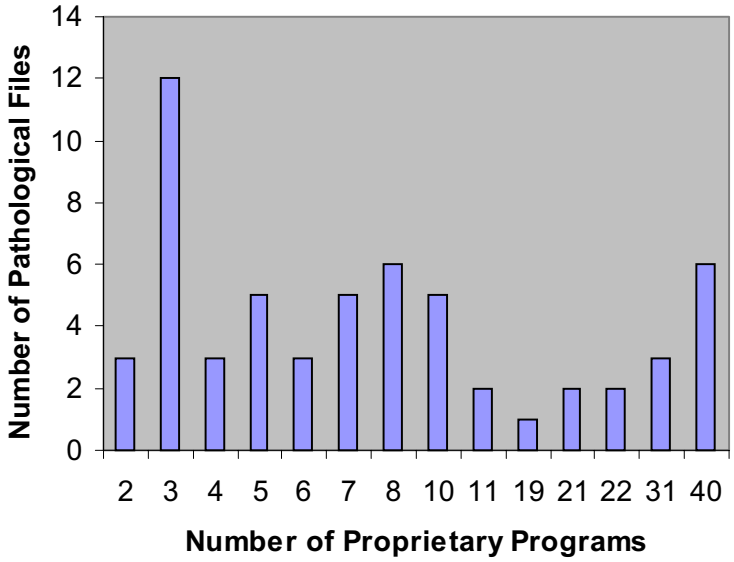

Figure 3. Distribution of the properties in the pathological files.

Table IV. Live data analysis.

Type of data	Cardinality before renewal	Cardinality after renewal
Live data	5403	4693
Nonredundant data	3825	4061
Redundant data	1578	632
Semantic data	946	0
Computational data	632	632

Unfortunately, the reverse engineering and restoration processes could not modify this situation, as this would have required redesigning the structure of the files and the programs that manage them.

One of the metrics confirming this high coupling among programs is the number of control data, which came to 558 on a total of 4693 (see Table IV). As has already been pointed out, these serve to create communication between programs through the files. Only reengineering could reduce their number.

There was also high coupling within the programs among components, which made the modules difficult to understand, as well as the relationships among them and therefore among the programs.

Copyright © 2001 John Wiley & Sons, Ltd.

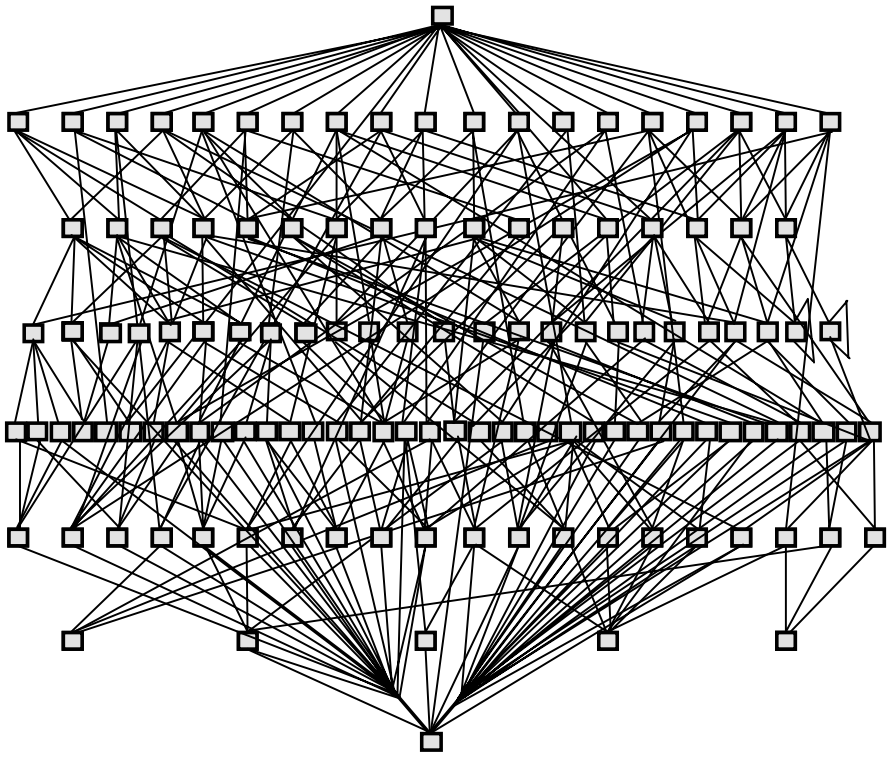

Figure 4. Call graph of program A0000 before restoration.

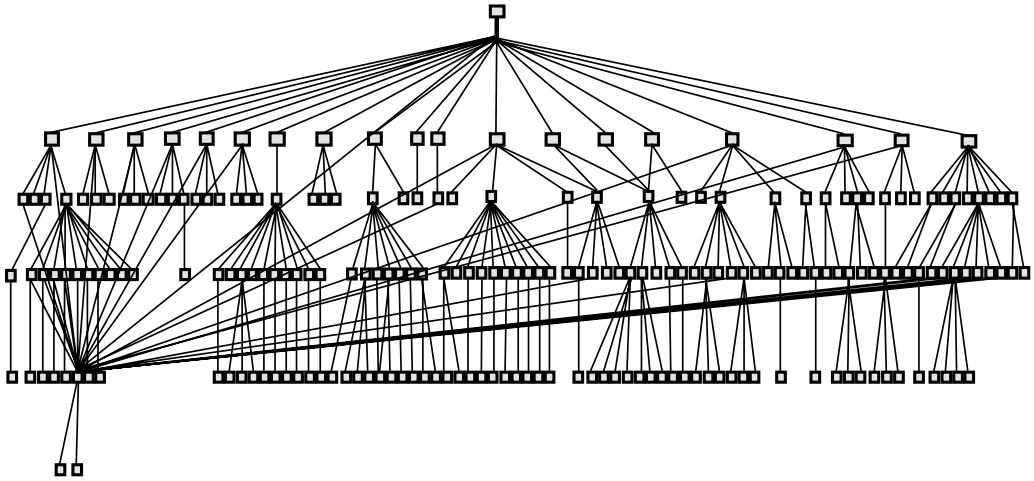

Figure 5. Call graph of program A0000 after restoration.

Copyright © 2001 John Wiley & Sons, Ltd.

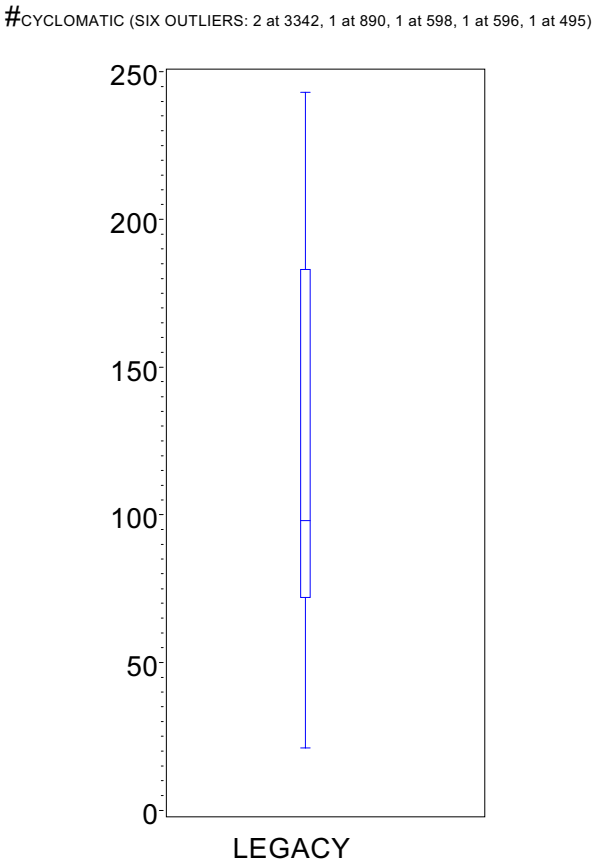

Figure 6. Cyclomatic numbers of the programs in the legacy system.

Figure 4 shows the call graph derived from a program named A000 that serves to collect, analyse and transmit to the relative programs all the transactions requested by the user. Figure 5 shows the same call graph after the renewal process; it should be noted that the result was generated entirely by the restoration process.

To quantify this coupling, Figure 6 shows the scatter plot of the cyclomatic numbers of the modules present in A0000. Approximately 75% of the total number of programs have cyclomatic numbers ranging from 70 to 170. The median value in all the programs in the system is 90.

Figure 7 shows the scatter plot of the cyclomatic numbers for the modules in the same program after restoration. Now, 25% of the modules have cyclomatic numbers below 3 and the other 75% do not exceed 12. The median value is around 7. Thus, the complexity of the modules can be seen to have greatly reduced, although some modules still have high complexity. The head of the renewal

Copyright © 2001 John Wiley & Sons, Ltd.

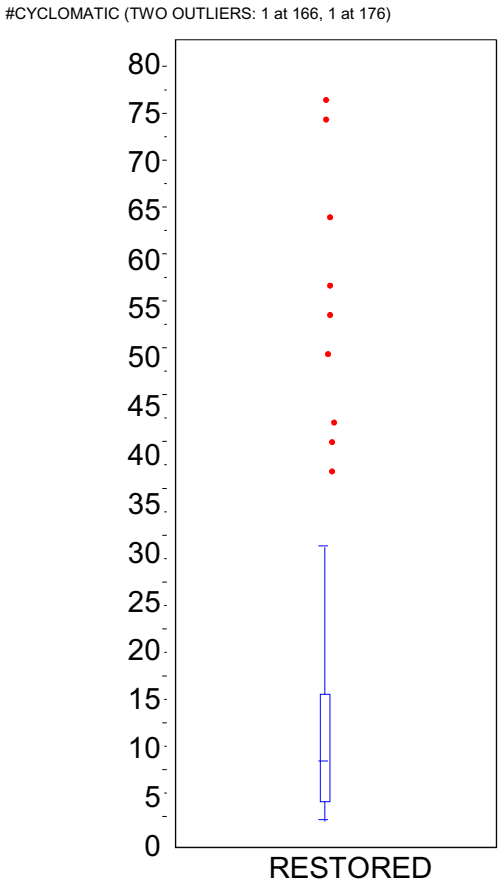

Figure 7. Cyclomatic numbers of the modules in the restored system.

project decided that renewing these modules would require too great an effort and so to have them rewritten, adapting their structure to the new architecture of the application which would result from the reengineering process.

Overall, there were 544 861 IF in the system, of which about 20% were algorithmic and the remaining 80% were procedural. Only about 50% (217 457) of these were eliminated, as the cost of the restoration activities for elimination of the others would have been too high.

The operations to treat this symptom are very costly and have a very critical trade-off.

Lesson 5. Coupling is one of the most harmful symptoms of ageing of a legacy system. Improvement is very costly and so it is one of the symptoms for which the renewal process should most definitely

Copyright © 2001 John Wiley & Sons, Ltd.

Table V. Analysis of the files.

Type of file	Cardinality before renewal	Cardinality after renewal
Useless	7	2
Obsolete	2	1
Temporary	2	2
Permanent	3	3
Anomalous	1	1
Pathological	57	57
Total	70	65

be anchored to the quality targets to be attained and to the resources available. Owing to its cost, it is better still to avoid the need for this process by preventive continuous monitoring of the corresponding metrics. In fact, if action is taken at the onset of the symptom, the trade-off will be more advantageous as the operations will have a more acceptable cost and be easier to carry out.

5.5. Layered architectures

Table V reports the measurements made on the files. The total number is not equal to the sum of the files in each class because some pathological files belong to different classes. The table shows that the renewal process produces a slight improvement in the metrics. It should be noted that among the 294 programs cancelled as useless, four managed one useless file each, that were also cancelled. Another useless file was eliminated during restoration. The remaining two were left in the legacy system because it would have cost too much to eliminate them as the programs that manage them are very complex.

Only one obsolete file was eliminated during restoration as for the others, again the effort required for their elimination would have been too high. The temporary, permanent and pathological files were unchanged after the renewal process. The full classification of the data collected in the files in this case study is shown in Table IV.

Thanks to the renewal process, the semantic redundancies were eliminated and the initial 946 data dropped to 236 unique items. This therefore increased the number of nonredundant data present after the renewal process. Instead, the computational redundancy could be measured but not improved except by reengineering.

The structural data were not altered by the renewal process because their reduction also requires reengineering. Their high number was due to the poor initial normalization of the database structure and the successive modifications, which made this metric progressively worse. There were 182 superimposed data structures, on a total of 65 files described. Those concerning the working data structures defined in each program were not measured.

Copyright © 2001 John Wiley & Sons, Ltd.

Table VI. Distribution of effort per process.

Process	Effort (%)
Measurement	10.41
Reverse engineering	24.20
Restoration	50.22
Equivalence test	15.17

The operations necessary to treat this symptom are not very efficacious and so it is better to monitor the legacy system carefully to observe and repair these before the efficacy of this treatment declines.

Lesson 6. Analysis of the metrics indicating the symptom of layered architecture is useful as a means of understanding why the initial design of the legacy system was not adequate to deal with evolutions in the application domain. This information can serve as a basis for improving the system architecture and the system design procedure during reengineering.

6. DISCUSSION

The previous sections show that in the renewal process, three types of activity may be carried out in both reverse engineering and restoration:

- automatic, performed by the software engineer using only the automatic tools he has available;
- reading, involving analyses of the software system and available documentation to extract information which cannot be explicitly read off, as well as the semantics of any concept which may aid comprehension;
- interviewing the users, maintainers or managers of the system to acquire the unstable information they possess.

In fact, the renewal process deals with two types of information:

- stable, i.e. described on a support which renders it unchangeable over time unless intentionally modified;
- unstable, i.e. the information possessed by human subjects, which, like human knowledge itself, is volatile and changes over time even if the reality it describes has not done so.

The process transforms all unstable information acquired in the interviews into stable information and thus makes the legacy system easier to read and understand, thereby also aiding knowledge transfer.

Table VI shows the distribution of the effort required for the activities selected for the software system under study.

The equivalence test served to verify that the system worked in exactly the same way after restoration. It was performed only after restoration because the structure of the programs was modified

Copyright © 2001 John Wiley & Sons, Ltd.

Table VII. Percentage of effort spent on the different methods.

Type of activity	Measurement (%)	Reverse engineering (%)	Restoration (%)
Automatic	65.45	48.32	23.57
Reading	23.37	14.08	67.21
Interviews	11.18	37.6	9.22

Table VIII. Productivity in reverse engineering and restoration.

Process	Program (LOC/h)	Data (data/h)
Reverse engineering	495.5	5.48
Restoration	201	2.17

during this subprocess, so that in fact, the effort spent should be summed with that spent on restoration. It has been counted separately because its cost does not depend on the age of the system like the restoration process itself, but on the quality of the test cases comprising the acceptance tests that the user expects to have carried out each time he receives a new version. Indeed, most of the effort required for this activity was due to collection of the case tests. The introduction of adequate technology for carrying out the collection of case tests from the exercise field, together with regression tests, could make the process more economical.

Table VII shows the relative distribution of the effort for the measurement activities and the renewal process. About half of the effort was supported by automatic tools in reverse engineering, so this was very productive in terms of work volume (see Table VIII). The automatic support in this phase was provided by static analysers which can reconstruct the documentation of the code and produce the information to be analysed. There is less reading in this phase than in restoration, because in the latter phase, a small part of the semantics has to be extracted, while most of the semantics is obtained through interviews. These are much more commonly held with users, maintainers and operators.

Restoration has far fewer automatic activities because the tools available are not yet mature enough to be able to realize the necessary slicing algorithms. There is a great deal of reading involved in this phase to extract the semantics necessary for restoring the application, whereas little time is spent on interviews because it uses stable information produced by reverse engineering. This table shows that restoration was the process that required the most effort, although it did not completely solve the problems requiring it. It shows the productivity achieved, analysed to give the reader an indication of the work volumes developed during the two phases of the analysis. The reference LOC and data are those of the legacy system before the renewal process; this shows that restoration reduces the productivity of the renewal process by about 50%.

Copyright © 2001 John Wiley & Sons, Ltd.

7. CONCLUSIONS

This study goes into some of the ageing symptoms of a legacy system. Each symptom has been specified by metrics, whose measurements suggest what operations should be carried out to renew the software. The expected efficacy of each operation is also described. The findings of this study can be used to help decide how to plan a renewal process and how best to manage it to increase its efficacy. For example, if the legacy system has programs or modules with large cyclomatic numbers, slicing tools and activities will be required during the restoration phase. This activity is very costly and the results are not likely to solve the problem entirely. It will therefore be necessary to decide on the depth of the restoration process to be carried out, making a trade-off between the quality improvement desired and the resources available.

Various lessons were learnt from the study; some suggested measures to be taken during the renewal process (Lessons 1 and 3), others indicated how to optimize the efficacy of the operations in the renewal process (Lessons 2 and 5). Some of the lessons demonstrated that prevention is better than cure (Lessons 4 and 5), while another suggested measurements that could be a basis for extracting perfective changes to be made to the system architecture (Lesson 6) if reengineering can be done.

In conclusion, the causes of ageing of a legacy system are as follows:

- poor quality of the configuration process;
- insufficient updating of the documentation after making changes to the code;
- inefficacious idiomatic practices;
- insufficient control of the software entropy;
- poor system design, that does not allow adaptation to variations in the application domain.

An additional finding besides those central to the research was that even when a legacy system shows no signs of ageing, it must be continually monitored to observe quality degradation at its onset and thus avoid the need for restoration procedures. The latter require more resources and do not entirely solve the problem if the symptom is disseminated in the legacy system . Unless these symptoms are subjected to timely treatment, terminal ageing of the software system will start to develop.

Some problems still left open by the present research, that should be addressed by the scientific community, are included in the following points:

- drawing up a more exhaustive list of symptoms and metrics, that should be completed by replicating this experiment in analogous conditions;
- reading techniques are very much used in renewal processes; as these must be continually carried out even on young systems to prevent premature ageing, such techniques should be refined to increase their accuracy and efficacy;
- the tools required to carry out renewal processes should achieve greater maturity to be able to improve both the efficacy and the efficiency of renewal processes, bearing in mind the permanent need for renewal processes together with ordinary maintenance;
- the reengineering process can solve the most critical problems of an aged system but it is difficult to apply this process because it cannot be carried out gradually like the restoration process. Reengineering processes that can be introduced gradually should studied in greater depth [29];
- the symptoms presented in this work are probably also applicable to systems with object-oriented (OO) architecture; as these, too, tend to age, it would be useful to verify whether such a transfer is possible and, if so, to adapt the concepts to OO systems.

Copyright © 2001 John Wiley & Sons, Ltd.

ACKNOWLEDGEMENTS

I am very much obliged to the company that gave me the opportunity to collect the experimental findings this work is based on. Thanks go to Ms M. V. C. Pragnell, for her aid as technical writer. Finally, I am very grateful to the anonymous referees for their helpful comments that have contributed to improve this work.

REFERENCES

1. Lehman MM. Laws of program evolution—rules and tools for programming management. *Proceedings of the Infotech State of the Art Conference*, April 1978. Pergamon Press, 1978; 11/1–11/25.
2. Visaggio G. Assessing the maintenance process through replicated, controlled experiments. *The Journal of System and Software* 1999; **44**:187–197.
3. Visaggio G. Value-based decision model for renewal process in software maintenance. *Annals of Software Engineering* 2000; **9**.
4. Chifosky EJ, Cross II JH. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990.
5. Abbattista F, Fatone GMG, Lanubile F, Visaggio G. Analyzing the application of a reverse engineering process to a real situation. *Proceedings Workshop on Program Comprehension*. IEEE Computer Society, 1994; 62–71.
6. Corbi TA. Program understanding: Challenge for the 1990s. *The Journal of Systems and Software* 1989; **28**(7).
7. Biggerstaff TJ. Design recovery for maintenance and reuse. *IEEE Computer* 1989.
8. Brown AJ. Specifications and reverse—engineering. *Software Maintenance: Research and Practice* 1993; **5**.
9. Cimitile A, Fasolino AR, Maresca P. Reuse reengineering and validation via concept assignment. *Proceedings Conference on Software Maintenance*, Montreal, Canada. IEEE Computer Society, 1993.
10. DeBaud JM, Rugaber S. A software re-engineering method using domain models. *IEEE Software* 1994.
11. Fanta R, Rajlich V. Removing clones from the code. *Journal of Software Maintenance: Research and Practice* 1999; **11**(4).
12. Lanubile F, Visaggio G. Extracting reusable functions by program slicing. *IEEE Transactions on Software Engineering* 1997; **23**(4).
13. Lehman MM, Perry DE, Ramil J. Implications of evolution metrics on software maintenance. *Proceedings Conference on Software Maintenance 1998*. IEEE Computer Society Press: Los Alamitos CA, 1998; 208–217.
14. Lehman MM, Perry DE, Ramil J, Turski WM, Wernick PD. Metrics and laws of software evolution—the nineties view. *Proceedings Symposium on Software Metrics 1997*. IEEE Computer Society Press: Los Alamitos CA, 1997; 20–32.
15. Ning JQ, Engberts A, Kozaczynski W. Automated support for legacy code understanding. *Communications of the ACM* 1994; **37**(5).
16. Oman PW, Cook CR. The book paradigm for improved maintenance. *IEEE Software* 1990; 39–45.
17. Ramil J, Lehman MM. Metrics of software evolution as effort predictors—a case study. *Proceedings Conference on Software Maintenance 2000*. IEEE Computer Society Press: Los Alamitos CA, 2000.
18. Solinger M, Engberts A, Ning JQ. Transferring re-engineering technology to a software development and maintenance organization: An experience report. *IEEE Software* 1994.
19. Welker KD, Oman PW, Atkinson GG. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice* 1997; **9**:127–159.
20. Ohlsson MO, Wohlin C. Identification of green, yellow and red legacy components. *Proceedings Conference on Software Maintenance*. IEEE Computer Society, 1998.
21. Clarke F *et al.* Subject-oriented design: Towards improved alignment of requirements, design and code. *Proceedings of OOPSLA '99*, 1999.
22. Kigzales G *et al.* Aspect-oriented programming. *Proceedings of ECOOP97*. Springer, 1997.
23. Lopes C *et al.* Aspect-oriented programming. *Proceedings of the Workshop ECOOP '99*. Springer, 1999.
24. Inhwan L, Ravishankar KI. Diagnosing rediscovered software problems using symptoms. *IEEE Transaction on Software Engineering* 2000; **26**(2).
25. Lanubile F, Visaggio G. Function recovery based on program slicing. *Proceedings Conference on Software Maintenance*, Montreal, Canada. IEEE Computer Society, 1993.
26. Cutillo F, Fiore P, Visaggio G. Identification and extraction of domain independent components in large programs. *Proceedings Conference on Reverse Engineering*, Montreal, Canada. IEEE Computer Society Press: Los Alamitos CA, 1993.
27. Visaggio G. Process improvement through data reuse. *IEEE Software* 1994; **11**(4).
28. Visaggio G. Assessment of a renewal process experimented on the field. *The Journal of Systems and Software* 1999; **45**(1).
29. Bisbal J, Lawless D, Wu B, Grimson J. Legacy information systems: Issues and directions. *IEEE Software* 1999; **16**(5).

Copyright © 2001 John Wiley & Sons, Ltd.

AUTHOR'S BIOGRAPHY

Giuseppe Visaggio graduated in Electronic Physics at the Universuty of Bari in 1972. After graduating, he continued to work in the same university in Computer Science and became Professor at its Informatics Department. His research interests are in maintenance, focusing particularly on processes, quality improvements and legacy systems. Currently he is Full Professor of Software Engineering at University of Bari. He is 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). Over the next three years he will serve the ICSM in the Steering Comittee. He is a member of the IEEE Computer Society, ACM and AICA (the Italian Computer Society).

Copyright © 2001 John Wiley & Sons, Ltd.