SOFTWARE DEFECT REPORT AND TRACKING SYSTEM IN THE INTERNET: CONTROLLING THE EVOLUTION OF LEGACY SYSTEMS

António Silva Monteiro[1], Miguel Afonso Goulão[2], Fernando Brito e Abreu[2], Alberto Bigotte de Almeida[1], Pedro Sousa[2]

Summary:
This paper describes a project of Empirical Software Engineering that uses Internet technology to implement a software defect report and tracking system, called SofTrack, in a geographically distributed organisation. SofTrack is being used to help controlling the evolution of four medium to large size software systems, on different levels of maturity. They belong to the Portuguese Navy Information Systems Infrastructure and were developed using typical legacy systems technology: COBOL with embedded SQL for queries in a Relational Database environment.
While SofTrack provides the typical functionality of tracking systems, it also embeds tools for the automatic documentation of source code and software complexity analysis. This combination of features helps the management of the maintenance team activity.

[1] email: damag@mail.marinha.pt
DAMAG - Portuguese Navy, Praça do Município 1188 Lisboa Codex, Portugal, ph.: 351.01.3468967, fax.: 351.01.3473154
[2] email: {miguel.goulao | fba | pms }@inesc.pt
Software Engineering Group, R. Alves Redol, 9, 13069 Lisboa Codex, Portugal, ph .: 351.01.3100306, fax.: 351.01.3145843

1. Introduction

Software Process Improvement (SPI) (Humphrey, 1990) is a mean of acquiring the ability to produce and maintain software of higher quality, at a lower cost and within schedule (Roberts, 1996), as it reduces the costs associated with poor software quality (Houston, 1996). DAMAG, the department responsible for the information systems infrastructure within the Portuguese Navy, is currently carrying out a SPI initiative.
Before the implementation of the actions described in this paper, most of the defects found, either in-house (by black box testing) or by final users, were simply handled on the telephone line. The majority of these actions were not recorded. Forecasts of short to medium term effort, required to support the software systems, were simply not available. This was particularly problematic since members of the teams in charge of the systems' evolution often had regular military assignments (e.g. naval exercises at sea) and those had to be planned with some advance. On the other hand, without on-line information of pending actions, the heads of the Software Division could not have control over the ongoing projects. Ultimately, the final users had no mechanism of feedback on their submitted requests.
To overcome this *status-quo* a DAMAG/INESC joint team was set up to define the architecture and develop a defect and reporting system that could be used in the expanding Navy Intranet. This team works as a Software Engineering Process Group (SEPG) in this initiative. The SofTrack (Software Defect Report and System Tracking) tool was then developed. SofTrack embeds a framework for classifying the requests.

SofTrack integrates the features of a typical web-based report and tracking system with the collection, analysis and presentation of software process and product complexity metrics. It also embeds a system for automatically generating documentation on the controlled software. SofTrack's web interface allows users spread across different buildings to have a customised view of the evolution of the systems they work with. This view is tailored to each user's activity in what regards that system, in order to make it easier for him to locate the relevant information for himself.
The paper presents some details on how all these features work together in the task of predicting and controlling the maintainability of a software system. A particular emphasis will be put on the support SofTrack gives to the software components complexity assessment, since it plays a very important role in the software maintenance complexity (Zuze, 1992).
This pilot experiment of Empirical Software Engineering is taking place in the Portuguese Navy Intranet for a few months. During this pilot stage of the SPI initiative, SofTrack is being used to track the evolution of four information systems (IS) built with the same products (COBOL and SQL/DS) in a proprietary system. They were developed by a different team and are now in different stages of their life cycle. The first IS was developed about a decade ago. The second one is being used since 1996. The other two systems are currently being tested and will be released to their final users during the first semester of 1999. All these systems are in continuous evolution, due to new user's requirements and external changes (such as the introduction of the Euro). The SEPG believes that this sample of IS provides a reasonable overview of the main defects detected in this kind of systems.

2. SofTrack

The conception and acceptance of SofTrack was not an easy nut to crack. It raised technological, methodological and cultural problems.
From a technological point of view, SofTrack combines the usage of commercial of the shelf software (COTS) with software developed by INESC and DAMAG. In short, SofTrack consists of a combination of tools to support the collection, analysis and report presentation of relevant product and process control information. This introduces changes in the methodological and cultural habits of the organisation subjects.
In essence, SofTrack uses three sources of data:

- The system's source code (product data).
- Requests for evolution actions, along with the corresponding follow-up and tracking data (process data).
- Other documents, such as product documentation, internal reports and scientific papers (miscelaneous). These are usually provided by the maintenance teams and the SEPG.

As a generic requisite, it was decided that all the interface with SofTrack's final users should be performed through the Portuguese Navy's Intranet. This only constrained the selection of tools in the sense that all the documents provided to final users must be produced in a suitable format (HTML) for any modern web browser.
SofTrack runs on a Windows NT web server with the Internet Information Server (IIS). The IIS is enhanced with the usage of a commercial add-on (ServletExec for Windows) to enable the support of Java servlets. Part of the data collection and software metrics computation processes is performed by a Unix workstation.

2.1. Architecture Overview

Figure 1 contains an overview of SofTrack's architecture.

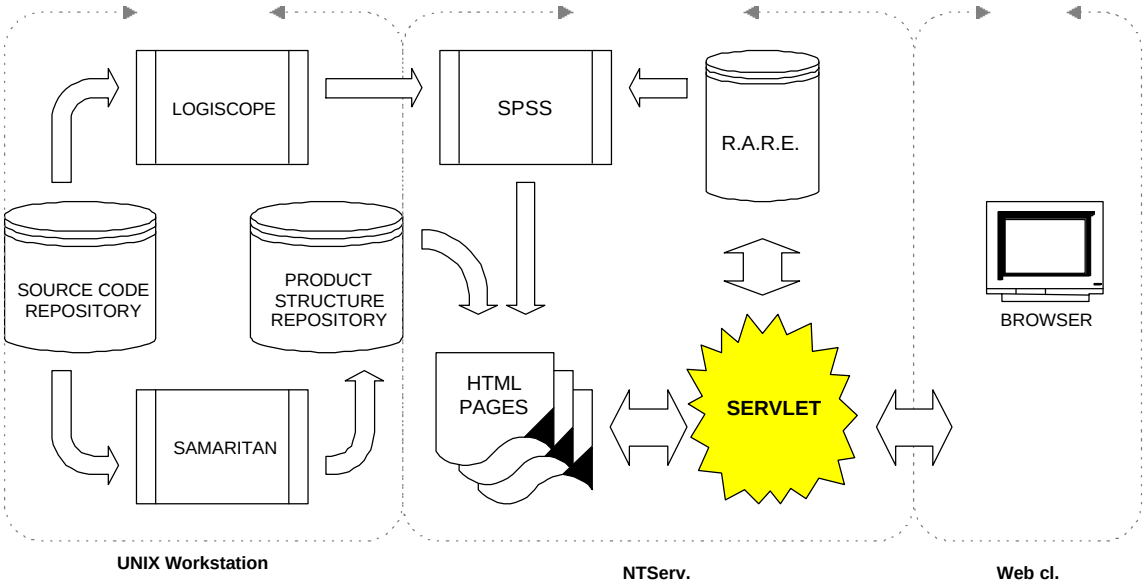

Figure 1. SofTrack architecture

Automated tools, that include COBOL parsers, perform the source code data collection. The source code files, stored in the *Source Code Repository*, are submitted to *Logiscope* (Verilog, 1993) and *Samaritan* (ESW, 1996), both running on a Unix workstation.

Logiscope is the COTS tool responsible for extracting data on the overall architecture of an application (Call Graph), the logical structure of its components (Control Graph) and measures of its complexity - software product metrics, such as the ones proposed in (Halstead, 1977) and (McCabe, 1976). *Logiscope* was chosen to perform this task because it supports the above stated features for the COBOL programming language (as well as for other languages). *Logiscope*'s outputs are the inputs for *Samaritan* and *SPSS* (Statistical Package for Social Sciences) (SPSS, 1997).

Samaritan is the tool responsible for the generation of updated documentation of the systems under analysis, based on their source code. The produced documentation is organised in the *Product Structure Repository* and presented in *HTML pages*, to allow an easy browsing through them.

The process data is collected with *RARE* (Register and Analysis of Requests of Evolution) (Goulão, 1998). In essence, *RARE* is a framework for storing information about evolution actions, from their request specification to the description of their follow-up, keeping track of the project's changed deliverables and the effort required to perform the described activities. RARE stores data in a format that is suitable for further statistical analysis. The entire user interaction with the database is performed through the web.

The *SPSS* tool is then used in the statistical analysis of the product metrics (collected with *Logiscope*), as well as the process data stored in the *RARE* database (this contains data for computing several process metrics). The *SPSS* scripting language makes it possible to schedule and execute a set of statistical procedures that can easily be expanded by adding new scripts. Moreover, these procedures are developed so that their output is in HTML format, making it simple to update SofTrack's web pages with a defined periodicity.

Finally, a java servlet is responsible for dealing with the interaction performed between clients and the SofTrack. In other words, this mixture of technologies works "behind the scenes", invisible to the final users. The servlet implements the data access policy of SofTrack, granting customised privileges to each user.

In the next sections we provide a few more details on the non-COTS components of

SofTrack.

2.2. Product structure - SAMARITAN

One of the problems the maintenance teams often have to deal with is the poor or outdated documentation of the maintained systems. Ironically, this makes the developers of the systems a very important asset, for most of the knowledge on how the system is implemented lies on their personal memories, instead of the documentation that should have been updated when the evolution actions were performed. This represents an even more serious problem when the original developers have to leave the maintenance team. In the particular case of the Portuguese Navy, this is a common issue, due to the military career of the members of the staff. All this makes the impact analysis of a requested evolution action more difficult.

While an organisation may implement a policy to ensure that from a given moment in time to the future, all changes will be documented, there is still the problem of documenting the current implementation of the IS.

Samaritan is a tool developed at INESC to perform this task. It analyses the system components and produces a dependency graph between physical artefacts such as executable images, source files, functions, databases, data base tables, and attributes of database tables.

Each node of the graph describes a single physical artefact. Each edge establishes a dependency between two artefacts. Since the Samaritan tool documents code and data artefacts, one must consider the following dependencies:

- *Create*, *read*, *update* and *delete* are possible dependencies between code and data artefacts.
- The *call* dependency is used to establish the control flow between functions.
- The *is-part-of* dependency is used to establish the organisation of code and data artefacts. For example, functions are part of source files and attributes are part of tables.

Some rules can be applied to the dependency graph, to reduce the number of links that must be maintained. For example, if function F *is-part-of* program P and F *read*s some database table attribute, then P also *read*s the same data, and no explicit link is necessary.

The graph is stored in the Samaritan internal repository and can be accessed for different purposes, such as complexity computation and impact analysis. In large information systems, the graph is often in the scale of hundreds of thousands of nodes and edges

For a better management of the dependency graphs' complexity and their full examination, Samaritan allows its users to generate customised sub graphs in a hypertext format, where nodes are HTML pages and edges are hypertext links between pages. These graphs may be navigated using any HTML browser.

The selection criteria to produce the sub graphs are based on the selection of the type of implementation artefacts and the type of dependency links. For instance, a user can produce a hypertext graph that describes the control flow, by selecting only artefacts of type Function and dependencies of type *call*.

The current version of Samaritan is able to parse source code in C and Cobol languages (in SUN, HP and IBM systems) with embedded SQL statements for ORACLE, INFORMIX, SQL/DS and DB2 databases. It also parses Oracle Stored procedures, Forms and triggers in PL/SQL language.

2.3. Evolution tracking – RARE

Tracking the evolution of a system gives the ability to pinpoint the evolution actions performed on a specific item, when and why they were performed, who was responsible for them and how much effort was spent on each of them. It also provides insight on the quality

of the tracked items and on the software process itself. In the context of SPI, the collection and analysis of software process metrics is a quantitative approach to the detection of flaws and pitfalls. It also allows the detection and validation of the best practices. Therefore, monitoring the evolution of a software system supports the SPI, facilitating the task of the maintenance teams.
The way the tracking data is recorded influences the type of analysis the SEPG may perform later. A normalised scheme for classifying the evolution requests (the RARE) was developed with the help of the maintenance teams, so that the scheme would be rich enough for the SEPG monitoring activities, yet, simple to use. The latter ensures that the maintenance teams don't have to spend too much time logging their activity (after a small training period, filling a RARE form takes no more than 5 minutes).
Both the requests for evolution actions (user environment) and their follow-up (maintenance team environment) are performed through the RARE web. Table 1 summarises the data contained in each RARE record.

User environment	Maintenance team environment
• Date • Identification of the evolution requester • Normalised description of the request • Textual description of the request • User's perception of the request's priority	• Identification of the responsible for the follow-up • Difficulty assessment • Normalised description of the evolution actions • Textual description of the evolution actions • Follow-up data (including relevant time-stamps) • Effort

Table 1. The RARE structure

Once submitted, the evolution requests are stored into the database for tracking purposes, and are then said to be in the "registered" stage. After an early assessment by the support department, the follow-up of the reported problem will be assigned to an appropriate software engineer.

2.4. Data access policy

One of the concerns of the common SofTrack user is to be able to quickly locate updated information that may be appropriate for the activity he is performing. On the other hand, the system contains classified information. For instance, there are reports on projects that should only be made accessible to that project's maintenance team and upper management, but not to the final users. This is clearly a problem of user profile definition.
The following user profiles were identified:

- The managers of the software process.
- The members of the maintenance teams (i.e., the programmers);
- The final users of the developed systems.

The actual definition of each profile's typical access rights was performed with the help of the SofTrack's users. The advantage of this approach was twofold: not only did it help to identify the main needs of information for each user profile, but it also made the final users feel more involved in the requirements definition of SofTrack. The second one holds the common

advantages that it brings on any application's acceptance by the final users.
The SofTrack architecture follows a more generic approach, for flexibility purposes. At a given point in time, the user will have access to the reports contained in the groups of reports of his access list and also to the reports accessible through the groups of users to which he belongs. Figure 2 contains a simplified example of the implementation of the access policy for a member of the maintenance team of Project A.

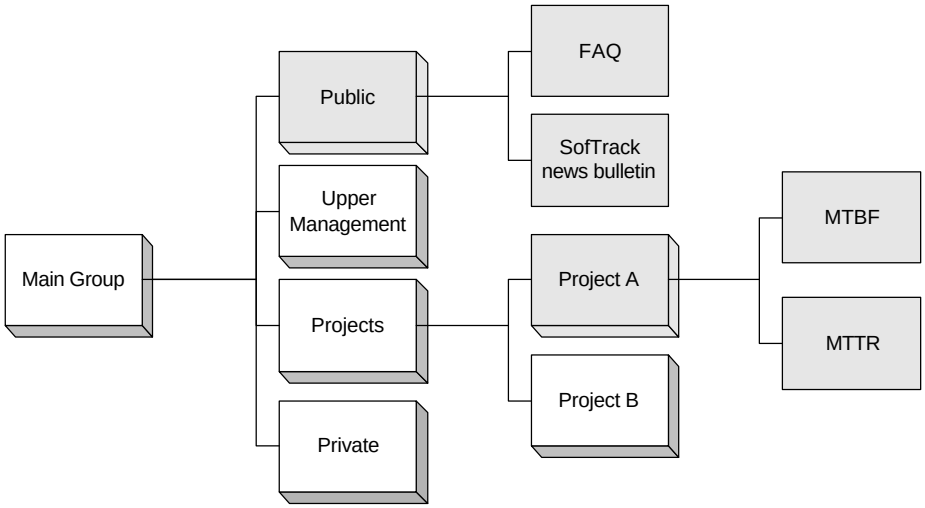

Figure 2. Example of user access privileges definition (only the grey ones are accessible).

After logging in the SofTrack, this user only sees the links to the Public pages and to Project A. The technical solution for this was the usage of a java servlet. The servlet enhances the HTTP server by enabling request/response services. This servlet interacts with the RARE database using a JDBC-ODBC bridge. Because all user information is passed to the servlet as part of the HTTP request, it implements additional authorisation as part of its service method.
When the user logs in the SofTrack, a cookie is set to carry his information inside the system. From here on, the whole environment is configured for that particular user. This means that the options presented to him are customised to his profile, keeping him from the unpleasant feeling of trying to follow a link and receiving back an error page because he does not have access to it. The servlet uses this cookie to dynamically generate the available links on the pages it serves back to the client.
Whenever the servlet receives the serve page request, it checks if the user has access to that URL and serves it back, providing he does have access. This double check on the accessibility of the page is performed to prevent users from composing an unauthorised page request.

3. Using SofTrack to assess maintainability

In the absence of an overall consensus within the software engineering community on what are the main aspects that define software quality, a good approach is to use the framework provided by ISO/IEC 9126 as a reference (ISO, 1995). Among the quality characteristics on that standard, one that is particularly relevant for this project is the maintenance complexity of the systems. In fact, one of the key concerns of SofTrack is to provide tools for detecting opportunities for making the maintenance activity simpler. This is achieved not only by the usage of automatic documentation of the source code and tracking of its evolution, but also by analysing process and product complexity metrics.

3.1. Process Analysis

The RARE is the basis for the tracking of the evolution of the software. The managers use it

to control the process. They can retrieve information about a particular action, or groups of actions. For instance, getting a Pareto chart on the frequency of evolution actions performed to each module helps to identify those modules that are more prone to changes (Figure 3), whether that derives from a poor specification or from a poor development. Since the motive of the change is registered, the chart may be further specialised to show the distribution for a particular one. For instance, one may see this same chart to check the frequency of evolution actions related to software bugs – the most frequent modules make up good candidates for further analysis and, eventually, reengineering actions.

Managers can also find out what are the most common motives for an evolution request (Figure 4). The codes in the horizontal axis of this chart are the ones used in the RARE framework. Here, the most common problem identified related to bugs in the output reports of the applications. Knowing the most typical defects in the software produced by the organisation helps defining the training policy, in order to suppress them.

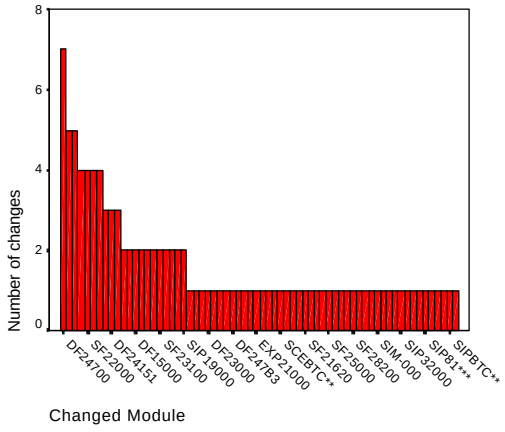

Figure 3. Module change proneness

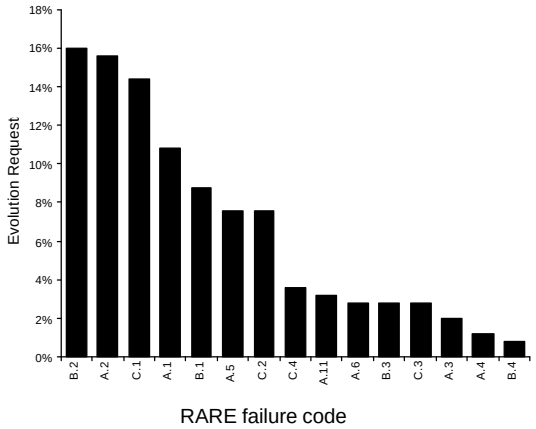

Figure 4. Evolution requests by failure type

The RARE database also allows the analysis of measures such as the medium time between failures (MTBF) and the medium time to repair (MTTR). The first one is a measure of the reliability of the software. A common usage of MTBF is to make a trend analysis of its value to decide when a software component is ready for release. MTTR gives an idea on how much time goes from the moment a request for change is submitted until it is fulfilled. It is an indirect measure of the maintenance difficulty of a software system.

3.2. Module complexity assessment and maintainability

The defect clustering phenomenon is relatively well known. The main reason for this is that, at least for medium to large systems, their complexity is far from being evenly distributed. Hence, by determining the overall complexity of the components, on may expect to find the most defect prone ones, since the defect clusters are usually located in the most complex software components.

But, how can one tell which are the most complex software components? There is no standard approach to answer this, in spite of the numerous efforts to assess software complexity. A survey by Zuze refered that more than 500 different complexity metrics had already been defined by 1991 (Zuze, 1992). Frequently, these metrics capture an aspect of the software complexity. So, one can use them individually to control that aspect. But to get an overview of what the overall complexity is, some authors such as (Khoshgoftaar, 1994) have chosen to combine several software complexity metrics. Assessing the complexity of a module based on a set of complexity measures is a somewhat similar problem to multicriteria decision making. Although we have several driving factors to make a judgement, in the end, we want to get an

overall value for complexity or a rank between the available options.
In this project, the SEPG tries to take advantage of single metrics to control specific issues of software complexity, but also combines them all to get an overall assessment.

3.2.1. Analysis of the evolution of development practises

The analysed systems were developed using similar techniques and development environment. As their main task is also the same – the manipulation of the personnel database, one could feel tempted to believe that there is no significant difference between the modules of different systems. To find out if this is true for our systems, we conducted an analysis of variance on the metrics collected in the modules of different systems that showed that there were significant differences between systems.
Table 2 presents a sample of the collected metrics' typical values for systems A and B, to illustrate how the comparison between both systems was performed. The outliers and extreme values of metrics were excluded. System A is currently in its testing phase, while system B has been in use for about a decade.

Table 2- Typical metrics values in different systems

	A			B		
	Mean	Std. Dev.	#mod.	Mean	Std. Dev.	#mod.
Direct Calls	14.6	9.0	1239	5.3	3.1	536
Maximum Levels	3.3	0.6	1239	6.5	2.8	536
Unconditional Jumps	0.0	0.4	1239	10.5	9.7	536
Statements	106.0	67.2	1239	105.1	59.4	536
Operands	262.4	176.6	1239	212.3	116.8	536
Operators	255.6	168.2	1239	218.9	120.3	536
Cyclomatic Complexity	22.7	13.7	1239	21.3	12.3	536

While some of the metrics present relatively similar values in modules of both systems, others show a completely different behaviour. For instance, the number of unconditional jumps is a lot lower in system A, indicating a change in the way the software is developed. This shows that although the technology employed to develop both systems is basically the same, some changes have occurred in the solutions chosen to implement them. This evolution is mostly due to the gap between the implementation of both systems, which amounts to almost a decade. The changes reflect the usage of the experience gathered with the older system when developing the more recent one.
The ISO/IEC 9126 standard describes maintenance complexity (or maintainability) through the following attributes:

- Analysability – potential effort spent in the diagnosis of defects and identification of the software items to be changed in an evolution action.
- Changeability – how hard it is to change software.
- Stability – when software is changed, this accounts for the eventual side effects.
- Testability – ability to validate the correctness of the changes performed.

The collected metrics are related to these attributes. For instance, McCabe's Cyclomatic Complexity (McCabe, 1976) is good measure for testability, as it reflects the number of separate paths one would have to test in order to test all the source code. The decrease in the typical maximum nesting level and on the number of unconditional jumps, along with the increase on the number of statements, operands and operators seems to indicate that a programming style trade-off was performed. The source code in project A is easier to

understand, with a smoother control structure (analysability). Interviews with the maintenance teams revealed that the increase in the number of direct calls results from a greater effort to separate the functionality of the system in more numerous but simpler modules. However, this might affect software stability, because the number of couplings grew (thus, possibly, so did the potential for side effects).

3.2.2. Overall complexity assessment

Normally, the most complex modules are more error prone, and because of that, more likely to require maintenance actions. Therefore, they make good candidates for analysis, in the event of a reengineering action. The rationale is that it is possible to reduce the overall system complexity if you focus on the small percentage of modules that are more complex. A short-term consequence of that complexity reduction is that it is possible to have smaller maintenance teams.

The information collected with RARE includes data on the effective effort spent in each particular evolution action and also on the changed modules, for each of those changes. Therefore, it is possible to track the changes down to the request that caused them and to find out how much time was spent with them. On the other hand, since we have the software versions before and after the evolution, we can evaluate the performed changes in terms of their complexity. We can use the variation of complexity to estimate the effort spent in the change.

As seen before, each collected metric measures an aspect of software complexity, but some of these aspects overlap (some of the collected metrics have a high correlation with each other). The collected metrics can be combined using the Principal Components Analysis technique (Reis, 1993; Kaiser 1958), so that we can get a reduced set of factors that hold the information on the variation of the complexity without a significant information loss. The derived metrics have a low correlation between them.

This technique allowed us to express the variation of complexity conveyed by the Logiscope metrics through three factors: F1 (program complexity, maximum nesting level, direct calls, exit nodes, entry nodes, maximum number of degrees, intelligent content, cyclomatic complexity, program level, program size, maximum number of nodes, maximum number of statements, nodes, edges, operators, operands and estimated number of errors), F2 (mental effort, unconditional jumps and essential complexity) and F3 (pending nodes).

These factors were used as estimators in the following linear regression model:

$$Effort_i = \beta_0 + \beta_1 * F1_i + \beta_2 * F2_i + \varepsilon_i$$

The β_j coefficients were computed by the least square method and ε_i represents the residual error for each case. It turned out that F3 had a neglectible effect on the registered effort ($\beta_3 \approx 0$), so, we decided to remove it from our complexity assessment model. This model may be instantiated with the parameters presented in Table 3.

Table 3 – Complexity assessment model parameters

Coefficient	Estimate	Standard Error	t statistic (5%)	Significance
β_0	6.682	0.470	14.221	0.000
β_1	3.104	1.060	2.928	0.006
β_2	8.522	0.587	14.508	0.000

$$Effort_i = 6.682 + 3.104 * F1_i + 8.522 * F2_i + \varepsilon_i$$

We may now examine briefly the model and check its statistical validity.

The positive coefficients show that an increase in any of the complexity factors results in the increase of the expected effort, as expected, due to the nature of the complexity metrics that were used to compute the factors.
The determination coefficient of the model (R^2) is 85.1%. Its adjusted value is 84.3%. This means that the model does not explain only 15.7% of the effort variation. For instance, the complexity of the database accesses is not accounted for, so, if a change involves modifying an embedded SQL statement or change the definition of a table, the model will underestimate the necessary effort. Including other measures to cover aspects such as the SQL complexity will likely enhance the model's ability to assess module complexity.
The F statistic value (108.561) with a significance of 0.05 is higher than the critical value for a sample of this dimension ($F_{(2, 38)} = 3.23$). This means we can reject the null hypothesis (H_0: $\beta_1 = \beta_2 = 0$).
We accounted for the efficiency of the computed predictors by performing the Durbin-Watson test and checking that there was no serial correlation of the residuals.
We also performed a Goldfeld-Quandt test, to check the homocedasticity of the variables of our model.
The number of observations used in the construction of this model is still considered by the metrics team a relatively low one. Only 41 evolution actions were considered. This makes the model vulnerable to a few factors. For instance, the model contains no information about the programmer who is responsible for implementing the evolution. (Brooks, 1975) refers an experiment performed by Sackman, Erikson and Grant (Sackman, 1968) where the authors show that the productivity of different programmers may differ drastically. In a group of experienced programmers, the best ones had a productivity level about 10 times better than the worst ones. The number of observations does not allow analysing effectively the productivity of the several programmers involved, but it might be necessary to weight the estimates in order to account for the programmer's productivity, in this model.
These aspects help explaining the current accuracy of the model in the prediction of effort. However, the model does give a good basis for assessing software component's complexity. As expected, tracing an overall complexity Pareto chart revealed that the core of the systems' complexity lies on a small percentage of their modules. Boxplot charts are used to select candidates for reengineering actions (those with extremely high values of complexity). Furthermore, for these candidates, analysing the individual complexity metrics helps understanding how they could be improved.

5. Conclusions

In this paper, the overall architecture of a system designed to support a SPI initiative was described. SofTrack is a system that combines several different technologies, from the information retrieval techniques used to extract data from the source code to the usage of web technology to make that information available to users in a distributed environment.
The standardisation of the way of requesting and tracking software evolution actions was a significant methodological and cultural change for the organisation where this SPI is taking place. It has created awareness to the need of producing software in a much more controlled environment within the organisation.
SofTrack provides an integrated view of most of the factors that contribute to software maintenance complexity:

- The Samaritan tool provides updated documentation on the software systems in a format that is suitable for quickly locating a software component and analysing its connections to the remaining system.
- The RARE framework provides a standard recording for all evolution actions and is the

basis for the evolution tracking system itself.

- The software complexity analysis uses a numerical approach to detect potential maintenance hot-spots, giving guidance for reengineering actions and for planning resource allocation for maintenance activities.

We believe this combination of features that are normally scattered across several systems holds in itself potential making the maintenance activity a much controllable one. Although this project is oriented for controlling the evolution of legacy systems, the basic principles are generic enough so that they may be applied to other systems.
For instance, if we were to take on a similar approach for tracking the evolution of OO systems, it would still be useful to have an architecture such as this one, although most of its specific components would have to be extended to accommodate this paradigm, or even replaced.

References

Brooks, F.: The mythical man-month. Addison-Wesley, 1975.
ESW: Samaritan – manual técnico. Internal Report - unpublished, 1996.
Florac, W.A., Park, R.E., Carleton, A.D.: Pratical Software Measurement: Measuring for Process Management and Improvement. CMU/SEI-97-HB-003, 1997.
Goulão, M., Monteiro,A., Martins, J., Bigotte de Almeida, A., Abreu, F.B., Sousa, P.: A software evolution experiment. ESCOM-ENCRESS98, Rome, 1998.
Halstead, M.: Elements of Software Science. Elsevier, North-Holland, New York 1977.
Houston, D., Keats, J.D.: Cost of Software Quality: A Means of Promoting Software Process Improvement. 1996.
Humphrey, W.S.: Managing the Software Process. SEI Series in Software Engineering, Addison-Wesley Publishing Company, 1990.
ISO: Information Technology - Software Quality Characteristics and Metrics. ISO/IEC, 1995.
Kaiser, H.F.: The Varimax Criterion for Analytic Rotation in Factor Analysis. Pshycometrica, 1958.
Khoshgoftaar, T.M., Munson, J.C., Lanning, D.L.: Alternative Approaches for the Use of Metrics to Order Program Complexity. Elsevier, 1994.
McCabe, T.: A Complexity Measure, IEEE Transactions on Software Engineering, Vol. 2, Nº4 pp 308-320, 1976.
Reis, E.: Análise factorial das componentes principais: um método de reduzir sem perder informação. Giesta ISCTE, 2ª Ed., 1993.
Roberts, M.A.: Experiences in Analizing Software Inspection Data. Proceedings of the Software Engineering Process Group Conference, 1996.
Sackman, H., Erikson, J., Grant, E.: Exploratory studies comparing online and offline programming performance. CACM, 11, 1, 1968.
SPSS: SPSS User's Guide. SPSS users manual package, 1997.
Verilog: Logiscope Viewer – Basic Concepts. Logiscope users manual package. Verilog SA., 1993.
Zuze, H.: Measuring Factors Contributing to Software Maintenance Complexity, 1992.
Zuze, H.: History of Software Complexity Metrics, 1992a.