LIFT: Reusing Knowledge from Legacy Systems

Kellyton dos Santos Brito[1, 2], Vinícius Cardoso Garcia[1, 2], Daniel Lucrédio[3], Eduardo Santana de Almeida[1, 2], Silvio Lemos Meira[1, 2]

[1]Informatics Center - Federal University of Pernambuco (UFPE)

[2]Recife Center for Advanced Studies and Systems (CESAR)

[3]ICMC - Institute of Mathematical and Computer Sciences - University of São Paulo (USP)

{ksb, vcg, esa2, srlm}@cin.ufpe.br, lucredio@icmc.usp.br

Abstract. *Software maintenance tasks are the most expensive activities on legacy systems life cycle, and system understanding is the most important factor of this cost. Thus, in order to aid legacy knowledge retrieval and reuse, this paper presents LIFT: a Legacy InFormation retrieval Tool, discussing since its initial requirements until its preliminary experience in industrial projects.*

1. Introduction

Companies stand at a crossroads of competitive survival, depending on information systems to keep their business. In general, since these systems many times are built and maintained in the last decades, they are mature, stable, with few bugs and defects, having considerable information about the business, being called as legacy systems [Connall 1993, Ulrich 1994].

On the other hand, the business dynamics demand constant changes in legacy systems, which causes quality loss and difficult maintenance [Lehman 1985], making software maintenance to be the most expensive software activity, responsible for more than 90% of software budgets [Lientz 1978, Standish 1984, Erlikh 2000]. In this context, companies have some alternatives: **(i)** to replace the applications by other software packages, losing the entire knowledge associated with the application and needing change in the business processes to adapt to new applications; **(ii)** to rebuild the applications from scratch, still losing the knowledge embedded in the application; or **(iii)** to perform application reengineering, reusing the knowledge embedded in the systems.

Reengineering legacy systems is a choice that prioritizes knowledge reuse, instead of building everything from scratch again. It is composed of two main tasks, Reverse Engineering, which is responsible for system understanding and knowledge retrieval, and Forward Engineering, which is the reconstruction phase. The literature [Lehman 1985, Jacobson 1997, Bianchi 2000] discusses several methods and processes to support reengineering tasks, as well as specific tools [Paul 1992, Müller 1993, Storey 1995, Finnigan 1997, Singer 1997, Zayour 2000, Favre 2001, Lanza 2003a, Lanza 2003b, Schäfer 2006] to automate it. However, even with these advances, some activities are still difficult to replicate in industrial context, especially in the first step

(reverse engineering) when there are a lot of spread information, sometimes with few or no documentation at all. Thus, tools that can aid and automate some of these activities are extremely essential. Despite the new tools available today some shortcomings still exist, such as: **(i)** the recovery of the entire system (interface, design and database), and trace the requirements from interface to database access, instead of only architectural, database or user interface recovery; **(ii)** the recovery of system functionality, i.e, ***what*** the system does, instead of recovering only the architecture, that shows ***how*** the system works; **(iii)** the difficult of managing the huge data amount present in the systems and the high dependency of the expert's knowledge; and **(iv)**, although existing tools address a proper set of requirements, such as search [Paul 1992], cognitive [Zayour 2000] or visualization capabilities [Schäfer 2006], they normally fail to address all the requirements together.

Thus, in this work, we present LIFT: Legacy InFormation retrieval Tool, which is a tool for knowledge retrieval from legacy systems, designed to fulfill the shortcomings identified in current tools. The remainder of this paper is organized as follows. In Section 2, we present the background of reengineering and reverse engineering, in order to clarify the terms and concepts used. In Section 3, we present the set of requirements of LIFT, based on a survey on reverse engineering tools, in conjunction with its architecture and implementation. Section 4 presents more details about the tool's functionality. In Section 5 we present a case study using the tool. Finally, in Section 6 we discuss some conclusions and future directions.

2. Background

According to the literature [Chikofsky 1990, Sommerville 2000, Pressman 2001], Reverse Engineering *is the process of analyzing a subject system to identify their components and interrelationships, in order to create representations in another form or at a higher abstraction level, as well as to recover embedded information, allowing knowledge reuse*. In this sense, reengineering is *the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*. In other words, reengineering is composed by a reverse engineering phase followed by a delta, which is reorganization or any alteration, and forward engineering.

The reengineering has basically four objectives [Sneed 1995, Bennett 2000]: **(i) to improve maintainability**: generating documents and building more structured, cohesive and coupled systems; **(ii) migration**: moving the software to a better or less expensive operational environment or convert old programming languages into new programming languages; **(iii) to achieve greater reliability**: the reengineering process has activities that reveal potential defects, such as re-documentation and testing; and **(iv) preparation for functional enhancement**: improving the software structure and isolating them from each other, make it easier to change or add new functions without affecting other modules. A great number of approaches and tools have been proposed to face the reengineering, and Garcia et al. [Garcia 2004, Garcia 2005] classified the main approaches in **(i)** *Source to Source Translation*, **(ii)** *Object recovery and specification*, **(iii)** *Incremental approaches* and **(iv)** *Component-based approaches*.

In reengineering, understanding the code is crucial to ensure that the intended behavior of the system is not broken. Studies show that these activities are the most

expensive tasks [Standish 1984, Erlikh 2000] and one of the efforts to decrease these costs is the development of software visualization tools [Bassil 2001], which make use of sophisticated visualization techniques to amplify cognition, and software exploration tools, which provide basic navigation facilities, i.e., searching and browsing [Robitaille 2000]. The boundary between these two categories is fuzzy, and Schäfer et al. [Schäfer 2006] defined an hybrid category, called visual exploration tools.

In this new category of visual exploration tools, Schäfer et al. [Schäfer 2006] discussed and justified five functional requirements and observed that existing tools only support a subset of them, in general each one covering different requirements. The requirements are: **(i)** *Integrated Comprehension*, **(ii)** *Cross-Artifact Support*, **(iii)** *Explicit Representation*, **(iv)** *Extensibilit,* and **(v)** *Traceability*.

Our work focuses on knowledge retrieval from legacy systems, based on reverse engineering, aiming to recognize and reuse the knowledge embedded in legacy systems, allowing the system maintenance or the forward engineering. We studied several software exploration and visualization tools, identifying their main requirements, in order to recognize not only five, but a solid set of initial requirements for a visual exploration tool. This study served as a basis for the definition of the LIFT tool, which is a tool for knowledge retrieval from legacy systems that complains to these identified requirements. In addition, we defined a new requirement that define techniques of automatic analysis and suggestions to user. Next section describes these requirements and the tool.

3. LIFT: Legacy Information Tool

In order to identify fundamental requirements for our visual exploration tool, we performed a survey covering the most known software tools with visualization and/or exploration capabilities. The survey included nine works: Scruple [Paul 1992], Rigi [Müller 1993], TkSee [Singer 1997], PBS [Finnigan 1997], SHriMP [Storey 1995], DynaSee[Zayour 2000], GSEE [Favre 2001], CodeCrawler [Lanza 2003a, Lanza 2003b] and Sextant [Schäfer 2006]. The identified requirements are shown in Table 1, and classified as follows:

R1. Visualization of entities and relations: Harel [Harel 1992] claimed that "*using appropriate visual formalisms can have spectacular effect on engineers and programmers*". An easy visualization of system modules and relationships, usually presented by a Call Graph, is an important issue of a software visualization tool, providing a graphical presentation of system and subsystems.

R2. Abstraction mechanisms and integrated comprehension: The visualization of large systems in one single view usually presents many pieces of information that are difficult to understand. Thus, the capability of presenting several views and abstraction levels as well as to allow user create and manipulate these views is fundamental for the understanding of large software systems.

R3. User interactivity: In addiction to the creation of abstractions and views, other interactivity options are also important, such as the possibility of the user take notes on the code, abstractions and views, which allows the recognition of information by another user or by the same user at some point again. This way, duplicated work can be

avoided. Another issue is the possibility of an easy switch between the high level code visualization and the source code, to permit the user to see both code representations without losing cognition information.

R4. Search capabilities: During software exploration, related artifacts are successively accessed. Thus, it is highly recommended to minimize the artifact acquisition time, as well as the number and complexity of intermediate steps in the acquiring procedure. In this way, the support of arbitrary navigation, such as search capabilities, is a common requirement in software reverse engineering tools.

R5. Trace Capabilities: Software exploration requires a large cognitive effort. In general, the user spends many days following the execution path of a requirement to understand it, and often it is difficult to mentally recall the execution path and the already studied items. Thus, to prevent the user from getting lost in execution paths, the tools should provide ways to backtrack the flows of the user, show already visited items and paths, and also indicate options for further exploration.

R6. Metrics Support: Visual presentations can present a lot of information in a single view. Reverse engineering tools should take advantage of these presentations to show some useful information in an effective way. This information can be metrics about cohesion and coupling of modules, length, internal complexity or other kinds of information chosen by user, and can be presented, for example, as the color, length and format of entities in a call graph.

R7. Cross artifacts support: A software system is not only source code, but a set of semantic (source code comments, manuals and documents) and syntactic (functions, operations and algorithms) information spread in a lot of files. So, a reverse engineering tool should be capable of dealing with several kinds of artifacts.

R8. Extensibility: The software development area is in constant evolution. The technologies and tools are in constant change, and their lifetime is even shorter. Thus, the tools must be flexible, extensible, and not technology-dependent, in order to permit its usage with a high range of systems and increase its lifetime.

R9. Integration with other tools: As software reuse researchers advocate [Krueger 1992], it is not necessary reinvent new solutions when others already exist, and a tool should permit that features present in other tools could be incorporated into it, as well as adopting standards to permit communication between distinct tools.

R10. Semi Automatic Suggestion: In general, the software engineer's expertise and domain knowledge are important in reverse engineering tasks [Sartipi 2000]. However, in many cases this expertise is not available, adding a new setback to system understanding. In these cases, the tool should have functionalities that automatically analyze the source code and perform some kind of suggestions to user, such as automatic clustering and patterns detection. However, we identified that this kind of requirement is not present in existent tools, and recognize it as a new requirement for knowledge recovery of software visualization and exploration tools.

3.1. Architecture

Based on the requirements defined in the previous section, we defined the LIFT architecture. In addition, we designed it to fulfill other non functional requirements,

such as Scalability, Maintainability and Reusability, and defined the main modules, the most important components, and expansion and integration points. The tool architecture is shown in Figure 1, and consists of four modules: *Parser*, *Analyzer*, *Visualizer* and *Understanding Environment*.

Table 1: Requirements of Software Visualization and Exploration Tools

Requirement	Tools								
	Scruple	Rigi	TkSEE	PBS	SHriMP	DynaSee	GSEE	Code Crawler	Sextant
Call Visualization		X	X	X	X		X	X	X
Abstraction Mechanisms		X			X				
User Interactivity	X	X		X	X		X	X	
Search Capabilities	X	X	X		X	X	X		
Trace Capabilities			X		X	X			X
Metrics Support		X						X	
Cross Artifacts Support		X	X	X					X
Extensibility		X	X	X					X
Integration			X	X		X			X
Semi Automatic Suggestion									

***Parser*:** It is responsible for organizing the source code. Thus, the legacy code is parsed and inserted in a structured database, in two steps: Initially, all code is parsed and stored in a first structure. Next, the parsed code is organized into a higher abstraction level, used by the application. This separation is useful to allow scalability, because the tool accesses the structure that contains only useful information, instead of all source code statements. The separation also allows an easy use of the toll with several technologies, since the use of a different input language can be made only by changing the parser component. Figure 2a shows some tables of the first parser and Figure 2b shows tables of the organized structure.

Analyzer: It is responsible for analyzing the code inserted in the structured database and to generate representations. First, by using pre-processing information, application modules are classified as interface and business one. In the first version of tool, this classification is based on language characteristics. For example, in NATURAL/ADABAS systems used in the case study, modules headers contain information if it is a map (interface module) or a program (business module). Next, the system database is inferred from legacy database instructions. Thus, a call graph is created with module information, such as size, type and source code comments.

Still within the analyzer, a second step is performed, to analyze and deduce the useful information. The tool uses mainly two methods: **(i)** minimal paths algorithm and **(ii)** cluster detection.

Minimal path is used to permit the user to follow the entire path from the user interface to data access. In this sense, the analyzer computes all minimal paths from all user interface and business modules to database entities, in order to support the user when following the system sequences. The minimal path implementation is based on the well-known Dijkstra algorithm [Dijkstra 1959].

On the other hand, cluster detection identifies and shows legacy system clusters that possibly can be recognized as a higher level abstraction, an object or component, or modules that can be merged to form one more cohesive structure. There are some approaches of cluster detection that are being used in reverse engineering context, mainly based on *k-means* algorithm [Sartipi 2000] that needs some previous knowledge about the code. We focus on unknown code, and choose the *Mark Newman's edge betweenness* clustering algorithm [Girvan 2002]. In this algorithm, the betweenness of an edge measures the extent to which that edge lies along shortest paths between all pairs of nodes. Edges which are least central to communities are progressively removed until the communities are adequately separated. We performed a small modification in the algorithm, which is the parameterization of the number of edges to be removed, allowing it to be interactively chosen by user.

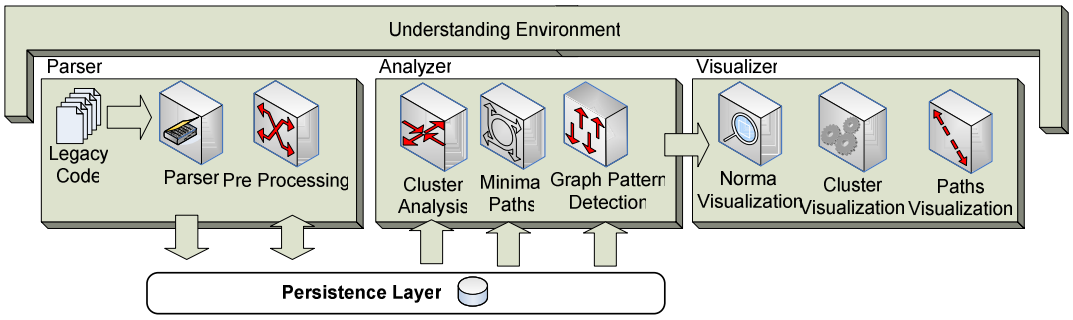

Figure 1: LIFT Architecture

Visualizer: It is responsible for managing the data generated by other modules, and to present these to the user in an understandable way. Visualization is based on a call graph, where modules and relationships are presented according to user preferences for modules and edges properties of thickness, color, format and size. The visualization has three modes, with easy transition among them: (i) default visualization, focusing on configurable attributes of modules and edges color, shape and size; (ii) path visualization, with focus on paths followed by application, with variable set of deep and direction (forward, upward or mixed mode) of paths; and (iii) cluster visualization, with focus on cluster detection and visualization.

Understanding Environment: Integrates the other modules, containing graphical interfaces for the functionalities of parser, code analyzer and visualizer. In addition, the environment provides an easy way to show the source code.

The tool works with the concept of code views. Thus, users can generate and deal in parallel with new sub graphs from previous graphs. The environment allows, for instance, the creation of graphs with only unconnected modules, which in general are dead code or batch programs. Other option is to generate new graphs with the detected clusters, isolating them from the complete application. These views are useful to isolate

modules and paths that identify application requirements.

The environment allows also the creation of system documentations, with areas to document the views, which represent a requirement, and areas to document the modules. The module documentation is created by default with source code comments, extracted in pre-processing. Additionally, the tool also permits the user to view and comment source code, maintaining both the original and the commented versions.

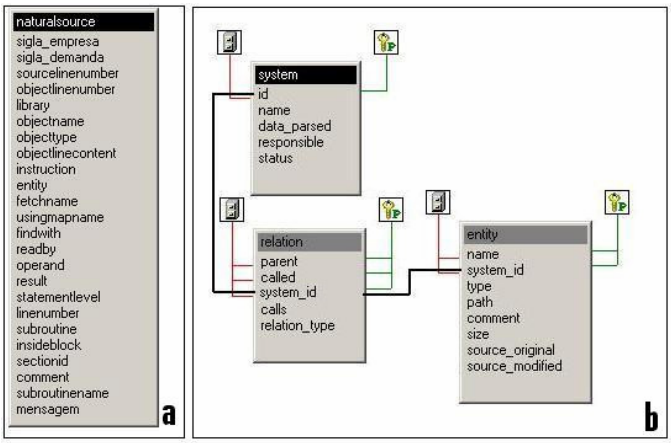

Figure 2: Database Tables used by the Parser

3.2. Implementation

In order to validate and refine the identified requirements and proposed architecture, we developed the LIFT tool in conjunction with the industry where we are engaged in reuse projects which focus on environments and tools to support reuse. These projects are part of the Reuse in Software Engineering (RiSE) project [Almeida 2004]. RiSE's goal is to develop a robust framework for software reuse, in conjunction with the industry, involving processes, methods, environment and tools. In the project, the role of the RiSE group (researchers) is to investigate the state-of-the-art in the area and disseminate the reuse culture. On the other hand, the industry (and its project managers, architects, and system engineers), represented by Recife Center for Advanced Studies and Systems (C.E.S.A.R[1]) and Pitang Software Factory[2], is responsible for "making things happens" with planning, management, and necessary staff and resources.

The LIFT was developed based on the architecture presented in previous subsection. The tool has a three-tier client-server architecture developed in JAVA.

The persistence layer uses SQL ANSI statements, therefore it is database independent. The parser was already implemented by the Pitang Software Factory as a .NET standalone application, and was refined and improved to be incorporated in LIFT. All other modules were developed in JAVA. Cluster analysis was developed based on *Mark Newman's edge betweenness* clustering algorithm and Minimal Paths was based

[1] Recife Center for Advanced Studies and Systems – http://www.cesar.org.br

[2] Pitang software factory – http://www.pitang.com

on *Dijkstra* algoritm, as shown in previous sections. The Visualizer uses the JUNG[3], Java Universal Network/Graph Framework, to implement visualizations.

The current version of LIFT implementation contains 76 classes, with 787 methods, divided into 25 packages, containing 9.420 lines of code (not counted code comments).

4. LIFT Usage

This section presents LIFT from a user's point of view. The initial steps, parsing, organization and call graph generation is performed by simple menu commands, shown in Figure 3a.

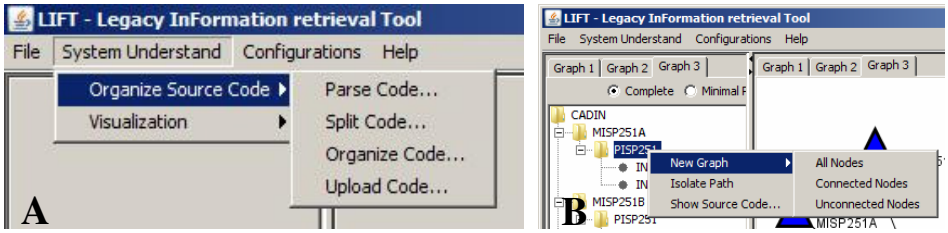

Figure 3. LIFT menus

The main screen, shown in Figure 4, has three areas. The left area (index 1) shows the paths and minimal paths from screens and business modules to database modules. The right area (index 2) shows the selected module information, such as the relations and comments, inserted by user or recognized by source code comments. In the center (index 3) the call graph is shown, with the tree visualization options (index 4).

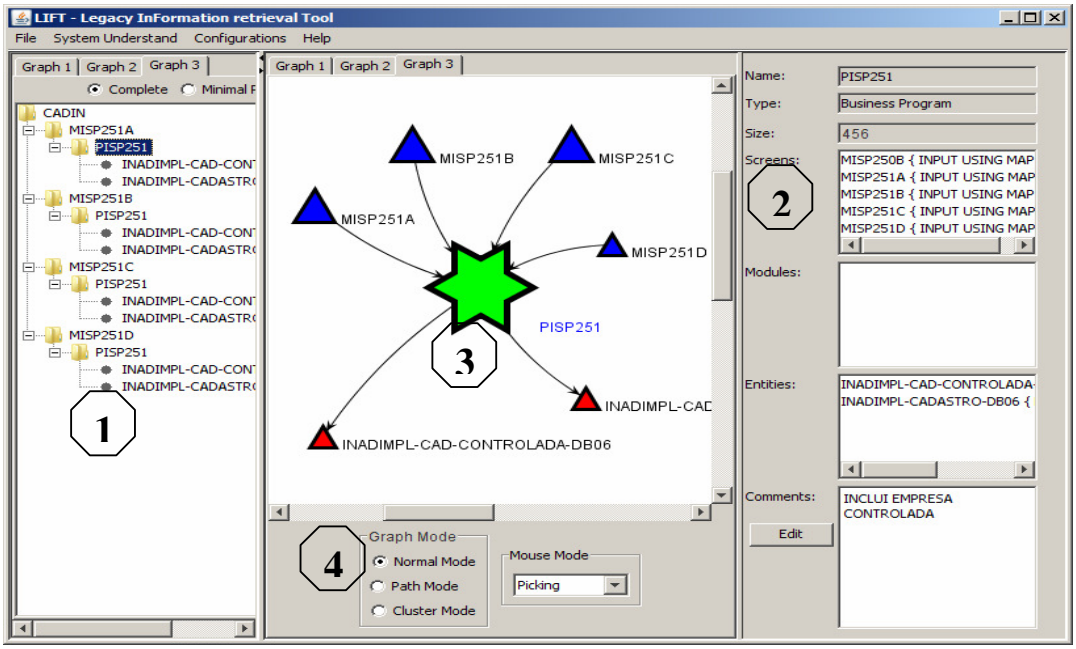

Figure 4: Isolated view in normal mode

[3] JUNG: Java Universal Network/Graph Framework - http://jung.sourceforge.net/

The first step towards system understanding is isolate unconnected nodes, which may be identified as dead code or batch programs. This task is performed by right clicking the paths area and choosing submenus “New Graph” and “Unconnected Nodes”, as shown in Figure 3b. These modules are analyzed separately from other modules. So, in a similar way, a new view containing only connected nodes is generated. In this view, the user tries to discover high coupled and related modules, by cluster detection, as shown in Figure 5a. Therefore, clustered modules are separated in a new view and analyzed in separate, in general resulting in a requirement. This new view is simpler than the complete view with all connected modules, providing an easier visualization of a possible requirement. Thus, by using the functionalities of path mode and analyzing the source code, the user can identify and generate documentation of the requirement. This documentation can be made in the *description area*, present in each view.

These steps are repeated until the entire application is separated into clusters, or no more clusters can be detected. In the last case, remaining modules are analyzed using the path mode (Figure 5b), in order to retrieve these requirements.

5. LIFT Preliminary Evaluation

LIFT is being used in a project involving C.E.S.A.R and the Pitang Software Factory. These institutions have acquired experience in understanding and retrieving knowledge from 2 million LOC of Natural/ADABAS systems, with programs varying from 11.000 LOC to 500.000 LOC. In these projects, only the source code is received and documents describing these are generated. Moreover, two kinds of knowledge retrieval can be performed: to understand for maintenance or understand for reimplementation in another platform.

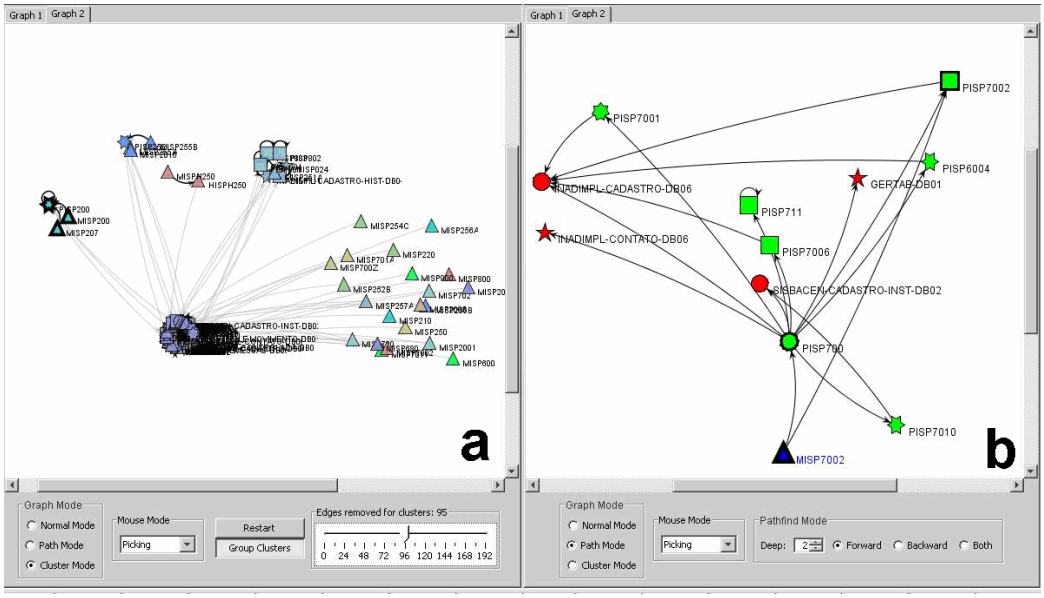

Figure 5: LIFT Cluster and Path Visualization Modes

Initially, LIFT was partially applied in a pilot project of understanding, for maintenance, a 65.000 LOC system. The understanding was performed in seven weeks,

by one Natural/ADABAS expert system engineer (SE) with no knowledge about analyzed code. The engineer recovered 10 high level requirements groups, each one with a proper set of sub requirements, which were validated by the project stakeholders. This pilot served to understand the company process, identify points where the tool could aid, refine it and improve its usability and scalability.

As a second interaction, the tool is being fully applied in a project of understanding for maintenance of a 210.000 LOC system. Initially, we made minor changes in the existing process to use the tool facilities. The modified process is shown in Figure 5 and defines four phases: **(i)** Organize Code, **(ii)** Analyze Code, **(iii)** Understand System and **(iv)** Validate Project. The three first phases are supported by LIFT. The new process is shown in Figure 5.

In this interaction, the source code was parsed and inserted into the database. Next, it was organized and the connected modules were separated from the unconnected ones and assigned for a system engineer for estimation and understanding. Thus, based on cluster analysis, SE experience and stakeholder information, the connected graph was split into 19 high level requirements groups, that would be responsible for each high level requirement. Next, based on the *Pitang staff* experience and company baseline, each group had a preliminary analysis and effort estimation. Thus, the understanding and documentation generation started.

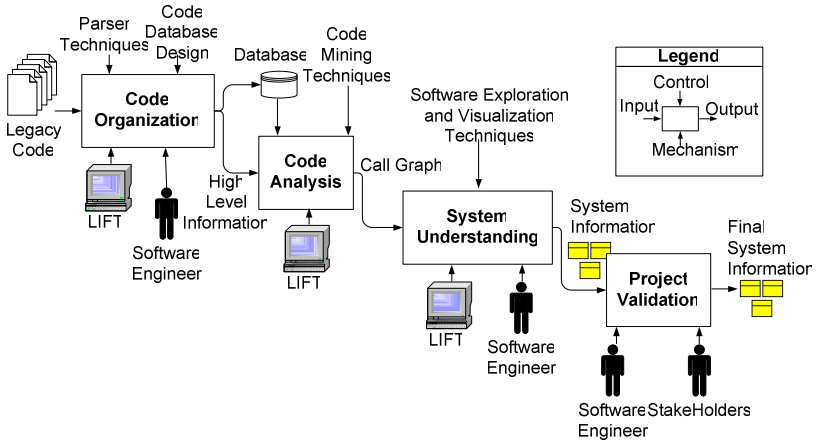

Figure 6: Process used to recovery legacy knowledge with LIFT

The requirements are documented as follows. A context diagram is generated, showing accesses to systems beyond application boundaries, such as access to other systems data or functions. Thus, *description area* comments of each view are mapped to functional requirements descriptions. Each functional requirement has also a tree with the modules and relationships of the correspondent view, that compound the modules and relationships of the requirement, and each one is described in details. In addition, each system data entity is recovered and described in another document section.

When this paper was being written, 12 of 19 high level requirements groups had already been understood. Table 2 shows estimated and executed understanding times of these groups. The real time to understand and document the groups was 38% less than the planned time, showing that the use of LIFT initially decreases the effort of knowledge retrieval tasks.

In addition to comparing estimation and executing time with the tool, we plan to compare final project data with previous project data, such as the number of requirements recovered, and total execution time by lines of code, function points and use case points. These data will be useful to validate these initial results.

Moreover, we are collecting user experiences to validate strong and weak points of the tool, as well as to identify possible new functionalities. Users agree that minimal paths visualization is very useful in knowledge recovery for re-implementation, because the key objective is to know the main application execution path, instead of details. However, the visualization of complete paths is desired in knowledge recovery for maintenance, because of the need for a map of the entire application when maintenance tasks are performed. Additionally, they agree that the use of views to isolate possible requirements and the existence of "*Path Mode*" are very useful to deal with large systems, allowing clean visualizations of large systems.

Another important consideration is that users reported that cluster analysis is useful to identify and isolate related modules, but the applicability of this option was limited when identifying the high level requirements groups because the NATURAL/ADABAS environment has some features that maintain and show to the user a list of the application entry and main modules. However, cluster analysis was useful to identify some of high level requirements not included on this list, and clusters and sub-requirements inside them.

Some points are being considered, as other forms of suggestions, such as text pattern detection in modules names, comments or source code, and document automatic generation.

Eventually, we ended up identifying that the tool can be used to understand large systems. Running in a Pentium IV/512MB Ram station and accessing a Pentium IV/512MB Ram database server, the tool performed tasks in the time shown in Table 3.

Table 2: Reduction in legacy knowledge recovery effort with LIFT

Group	Estimation(h)	Execution(h)	Gain
Group 1	13,00	11,00	15%
Group 2	10,00	8,00	20%
Group 3	18,00	11,00	39%
Group 4	13,00	8,00	38%
Group 5	10,00	6,00	40%
Group 6	2,00	1,00	50%
Group 7	18,00	9,00	50%
Group 8	18,00	8,00	56%
Group 9	10,00	6,50	35%
Group 10	7,00	5,00	29%
Group 11	1,00	1,00	0%
Group 12	1,00	1,00	0%
TOTAL	**121,00**	**75,50**	**38%**

Parse Code and *Organization* are slow tasks, but we consider that this time does not harm the tool's performance because these tasks occur only once in each system. In addition, *Minimal Paths Calculation*, *Analysis* and *Load Graph* tasks take a small time, but are performed few times, that is, only when the application runs and the system is chosen. Besides, *Cluster Detection* is a task that takes little time, in general from 1 to 20 seconds depending on the number of modules and edges involved. Finally, after initial

analysis, the operations of *graph manipulation*, *view creations* and *load details* are instant tasks, with times imperceptibles by user, which provides a good usability and user experience.

6. Concluding Remarks and Future Work

In this paper, we presented LIFT: Legacy InFormation retrieval Tool, which is a tool for knowledge retrieval from legacy systems. The tool requirements were obtained from a set of functionalities of other tools for software visualization or exploration, from our industrial experience and customer needs, as well as a new requirement was identified as a lack on current tools, that is the analysis of source code and the perform of suggestions to user. The tool uses the concept of views, which encloses the system functionalities. Additionally, we also use a new way to store legacy data in database systems, which allows tool scalability.

The tool was partially used in a pilot project to knowledge retrieval of a 65.000 LOC NATURAL/ADABAS system, in order to refine and improves its usability and scalability. Currently, the tool is being fully used in another similar project of software understanding and knowledge recovery for the maintenance of 210.000 LOC. By the end of the second understanding interaction, we intend to compare data with previous projects, in order to get more accurate information and make better conclusions about the effort reduction.

Table 3: LIFT execution times

Task	Time (seconds)	
	65KLOC Application	210 KLOC Application
Parse Code	961s	1520s
Organize Code	660s	963s
Minimal Paths Calculation	19s	84s
Full Analysis and Graph Creation	23s	98s
Interactive Cluster Detection	0s - 20s	0s – 30s
Graph Manipulation	Imperceptible	Imperceptible

The preliminary results show effort reduction of 38% in relation to the estimated time. Also, user interviews demonstrate best applicability of minimal paths in tasks of general understand for re-implementation, and complete paths in tasks of full understanding for maintenance. We observed good user experiences in knowledge recovery using the concepts of requirements isolation in multiple views, cluster detection and *path mode* system navigation. Thus, the tool presents good scalability performing tasks in user acceptable times, running in conventional hardware.

Currently, LIFT is being modified to perform automatic documentation generation: textual requirements document and UML diagrams, such as use case and class diagrams. Also, general and graph pattern detection are being implemented, in order to provide more suggestions to user and increase the effort reduction. Finally, we plan to use the tool to perform reverse engineering and knowledge recovery in systems that use other technologies or development paradigm, such as COBOL or Java.

References

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C. and Meira, S. R. d. L. (2004). "RiSE

Project: Towards a Robust Framework for Software Reuse". IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA, p. 48-53.

Bassil, S. and Keller, R. K. (2001). "Software Visualization Tools: Survey and Analysis". Proceedings of International Workshop of Program Comprehension, Toronto, Ont., Canada, p. 7-17.

Bennett, K. H. and Rajlich, V. T. (2000). "Software maintenance and evolution: a roadmap". Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, ACM Press, p. 73-87.

Bianchi, A., Caivano, D. and Visaggio, G. (2000). "Method and Process for Iterative Reengineering of Data in a Legacy System". Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, IEEE Computer Society, p. 86--97.

Chikofsky, E. J. and Cross, J. H. (1990). "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software Vol.(1), No. 7, p. 13-17.

Connall, D. and Burns, D. (1993). "Reverse Engineering: Getting a Grip on Legacy Systems." Data Management Review Vol.(24), No. 7.

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." Numerische Mathematik Vol.(1), No. 1, p. 269-271.

Erlikh, L. (2000). "Leveraging Legacy System Dollars for E-Business." IT Professional Vol.(2), No. 3, p. 17-23.

Favre, J.-M. (2001). "GSEE: a Generic Software Exploration Environment". Proceedings of the International Workshop on Program Comprehension (IWPC), Toronto, Ont., Canada, p. 233.

Finnigan, P. J. (1997). "The software bookshelf." IBM Systems Journal Vol.(36), No. 4.

Garcia, V. C. (2005), "Phoenix: An Aspect Oriented Approach for Software Reengineer(in portuguese). M.Sc Thesis." Federal University of São Carlos, São Carlos,

Garcia, V. C., Lucrédio, D., Prado, A. F. d., Alvaro, A. and Almeida, E. (2004). "Towards an effective approach for reverse engineering". Proceedings of 11th Working Conference on Reverse Engineering (WCRE), Delft, Netherlands, p. 298-299.

Girvan, M. and Newman, M. E. J. (2002). "Community Structure in Social and Biological Networks." Proceedings of the National Academy of Sciences of USA Vol.(99), No. 12.

Harel, D. (1992). "Toward a Brighter Future for System Development." IEEE Computer Vol.(25), No. 1.

Jacobson, I., Griss, M. and Jonsson, P. (1997). "Software Reuse: Architecture, Process and Organization for Business Success", Addison-Wesley Professional.

Krueger, C. W. (1992). "Software Reuse." ACM Computing Surveys Vol.(24), No. 2, p. 131-183.

Lanza, M. (2003a). "CodeCrawler - lessons learned in building a software visualization tool". Proceedings of European Conference on Software Maintenance and Reengineering, p. 409-418.

Lanza, M. and Ducasse, S. p. (2003b). "Polymetric Views-A Lightweight Visual Approach to Reverse Engineering." IEEE Transactions on Software Engineering Vol.(29), No. 9, p. 782-795.

Lehman, M. M. and Belady, L. A. (1985). "Program Evolution Processes of Software Change", London: Academic Press.

Lientz, B. P., Swanson, E. B. and Tompkins, G. E. (1978). "Characteristics of Application Software Maintenance." Communications of the ACM Vol.(21), No. 6, p. 466 - 471.

Müller, H. A., Tilley, S. R. and Wong, K. (1993). "Understanding software systems using reverse engineering technology perspectives from the Rigi project". Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, p. 217-226.

Paul, S. (1992). "SCRUPLE: a reengineer's tool for source code search ". Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, IBM Press, p. 329-346

Pressman, R. S. (2001). "Software Engineering: A Practitioner's Approach", McGraw-Hill.

Robitaille, S., Schauer, R. and Keller, R. K. (2000). "Bridging Program Comprehension Tools by Design Navigation". Proceedings of International Conference on Software Maintenance (ICSM), San Jose, CA, USA, p. 22-32.

Sartipi, K., Kontogiannis, K. and Mavaddat, F. (2000). "Architectural design recovery using data mining techniques". Proceedings of the 4th European Software Maintenance and Reengineering (ESMR), Zurich, Switzerland, p. 129-139.

Schäfer, T., Eichberg, M., Haupt, M. and Mezini, M. (2006). "The SEXTANT Software Exploration Tool." IEEE Transactions on Software Engineering Vol.(32), No. 9.

Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N. (1997). "An examination of software engineering work practices". Proceedings of conference of the Centre for Advanced Studies on Collaborative research (CASCON), Toronto, Ontario, Canada, IBM Press, p. 21.

Sneed, H. M. (1995). "Planning the Reengineering of Legacy Systems." IEEE Software Vol.(12), No. 1, p. 24-34.

Sommerville, I. (2000). "Software Engineering", Pearson Education.

Standish, T. A. (1984). "An Essay on Software Reuse." IEEE Transactions on Software Engineering Vol.(10), No. 5, p. 494-497.

Storey, M.-A. D. and Müller, H. A. (1995). "Manipulating and documenting software structures using SHriMP views". Proceedings of the International Conference on Software Maintenance (ICSM), Opio, France, p. 275 - 284.

Ulrich, W. (1994). "From Legacy Systems to Strategic Architectures." Software Engineering Strategies Vol.(2), No. 1, p. 18-30.

Zayour, I. and Lethbridge, T. C. (2000). "A cognitive and user centric based approach for reverse engineering tool design". Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research, Ontario, Canada, p. 16.