Journal of Universal Computer Science, vol. 14, no. 8 (2008), 1256-1284
submitted: 15/11/07, accepted: 25/3/08, appeared: 28/4/08 © J.UCS

LIFT – A Legacy InFormation Retrieval Tool

Kellyton dos Santos Brito
(Recife Center for Advanced Studies and Systems, Federal University of Pernambuco, Brazil
kellyton.brito@cesar.org.br)

Vinícius Cardoso Garcia
(Recife Center for Advanced Studies and Systems, Federal University of Pernambuco, Brazil
vinicius.garcia@cesar.org.br)

Eduardo Santana de Almeida
(Recife Center for Advanced Studies and Systems, Brazil
eduardo.almeida@cesar.org.br)

Silvio Romero de Lemos Meira
(Recife Center for Advanced Studies and Systems, Federal University of Pernambuco, Brazil
silvio@cesar.org.br)

Abstract: Nowadays software systems are essential to the environment of most organizations, and their maintenance is a key point to support business dynamics. Thus, reverse engineering legacy systems for knowledge reuse has become a major concern in software industry. This article, based on a survey about reverse engineering tools, discusses a set of functional and non-functional requirements for an effective tool for reverse engineering, and observes that current tools only partly support these requirements. In addition, we define new requirements, based on our group's experience and industry feedback, and present the architecture and implementation of LIFT: a Legacy InFormation retrieval Tool, developed based on these demands. Furthermore, we discuss the compliance of LIFT with the defined requirements. Finally, we applied the LIFT in a reverse engineering project of a 210KLOC NATURAL/ADABAS system of a financial institution and analyzed its effectiveness and scalability, comparing data with previous similar projects performed by the same institution.

Keywords: Reverse Engineering, Knowledge Reuse, System Understanding, Legacy Systems
Categories: D.2.1, D.2.7, D.2.13, K.6.3

1 Introduction

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

On the other hand, business dynamics demand constant changes in legacy systems, which causes quality loss and difficult maintenance [Lehman and Belady, 1985], making software maintenance to be the most expensive software activity, responsible for more than 90% of software budgets [Lientz et al., 1978, Standish, 1984, Erlikh, 2000]. In this context, companies have some alternatives: **(i)** to replace the applications with other software packages, losing the entire knowledge associated with the application and needing changes 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 and Belady, 1985, Jacobson et al., 1997, Bianchi et al., 2000] discusses several methods and processes to support reengineering tasks, as well as specific tools [Paul, 1992, Müller et al., 1993, Storey and Müller, 1995, Finnigan, 1997, Singer et al., 1997, Zayour and Lethbridge, 2000, Favre, 2001, Lanza, 2003, Schäfer et al., 2006] to automate it. However, even with these advances, some activities are still difficult to replicate in industrial contexts, especially in the first step (reverse engineering) where a huge amount of information is spread, sometimes with few or no documentation at all. Thus, tools that can aid and automate some of these activities are extremely essential.

However, even with the tools available today some flaws still exist, such as the difficulty of managing the huge data amount present in the systems, the recovery of systems functionalities, instead of recovering only the architecture, and the dependency of the expert's knowledge.

In this context, this work defines the requirements, designs and implements a tool for reverse engineering, aiming to aid system engineers to retrieve knowledge from legacy systems, as well as to increase their productivity in reverse engineering and system understanding tasks. Moreover, the tool is based on the-state-of-the-art and practice in the area, and its foundations and elements are discussed in details.

In a previous works we introduced a preliminary version of the tool [Brito et al., 2007b] and presented its first version for community analysis in a conference's tool session [Brito et al., 2007a]. This paper makes two novel contributions:

- A more detailed discussion about the requirements for a robust reverse engineering tool.

- A case study using the tool in an industrial context of reverse engineering a 210 KLOC NATURAL/ADABAS system of a financial institution, comparing the results with two similar projects of reverse engineering systems with 65 KLOC and 210 KLOC, performed without the use of LIFT tool.

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, the main approaches and future trends. In Section 3, we briefly present a survey about reverse engineering tools, and the set of requirements for a robust tool, based on this survey. Section 4 presents the LIFT – Legacy InFormation retrieval Tool, based on the requirements presented in Section 3, in addition to new requirements based on the flaws of current research. In addition, the architecture, implementation and an example of tool usage are also presented. Section 5 discusses the case study. Finally, in Section 6 we discuss some conclusions and future directions.

2 Reengineering and Reverse Engineering

According to research [Chikofsky and Cross, 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 addition, **Forward Engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system, following a sequence from requirements through designing its implementation. Finally, **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.

Since the initial research on software maintenance and reengineering [Lientz et al., 1978], several approaches were proposed trying to automate or aid software engineering in their activities. Garcia et al. [Garcia et al., 2004, Garcia, 2005] performed an extensive study of these approaches, identifying four lines: **(i)** Source-to-Source Translation, **(ii)** Object Recovery Specification, **(iii)** Incremental Approaches and **(iv)** Component Based Approaches.

Source-to-Source Translation: The source program first transliterated into the target language on a statement-by-statement basis. Refinements are then applied in order to improve the quality of the output, mainly because it to be insufficiently sensitive to global features of the source program and too sensitive to irrelevant local details.

Object recovery and specification: The idea of applying object-oriented reverse engineering provides a simple way to create models of an existing system. The Object-Oriented paradigm offers some desirable characteristics, which significantly help in improving software reuse.

Incremental approaches: There are several benefits associated with iterative processes: by using “divide et impera” (“divide-to-conquer”) techniques, the problem is divided into smaller units, which are easier to manage; the outcomes and investment return are immediate and concrete and the risks associated to the process are reduced, among other benefits.

Component-Based approaches: The extraction of reusable software components from an entire system is an attractive idea, since software objects and their relationships incorporate a large amount of experience from past developments. It is necessary to reuse this experience in the production of new software.

In addition, new trends can be recognized in the reengineering area, such as **Aspect Oriented approaches** [Garcia et al., 2005], which try to identify possible crosscutting concerns from the source code and extract them through refactorings into new aspect-oriented code. Furthermore, **Data Mining approaches** [Sartipi et al., 2000, El-Ramly et al., 2002] focus on recovering legacy systems knowledge by mining them in databases.

Even with the existence of many processes, methods and approaches to reengineering, some flaws still exist. Currently, unresolved issues include **(i)** the recovery of the entire system (interface, design and database), and to 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 **(iv)** the high dependency of the expert's knowledge.

In addition, studies trying to establish a roadmap for reengineering and reverse engineering research for the new millennium [Müller et al., 2000, Canfora and Penta, 2007] identified that tool integration and adoption should be central issues for the next decade. Also, it is necessary to evaluate reverse engineering tools and technology in industrial settings with concrete reengineering tasks at hand, to increase tool maturity and interoperability, and this adoption.

3 Reverse Engineering Tools

Despite the maturity of reengineering and reverse engineering research, and the fact that many pieces of reverse engineering work seem to timely solve crucial problems and to answer relevant industry needs, studies [Müller et al., 2000, Canfora and Penta, 2007] indicate that the adoption of current available tools to automate the tasks in industry is still limited.

In this regard, we surveyed the state-of-the-art and practice on the reverse engineering tools field, trying to establish some relations between them, in order to define a base for the requirements of an efficient reverse engineering tool. The survey was based on the main literature of reengineering, reverse engineering and software engineering areas, including the *Working Conference on Reverse Engineering (WCRE)*, the *International Conference on Software Maintenance (ICSM)*, the *European Conference on Software Maintenance and Reengineering (CSMR)*, the *International Conference on Program Comprehension (ICPC)*, the *International Conference on Software Engineering (ICSE)*, the *IEEE Transactions on Software Engineering*, and the *Journal of Systems and Software*, among others. In addition, web search engines, such as www.scirus.com and www.google.com, and the web portal of ACM and IEEE organizations were also consulted, aiming to find more data related to the problem. The survey covered the most known tools, including eight works: Scruple [Paul, 1992], Rigi [Müller, 1993], TkSee [Singer, 1997], SHriMP [Storey, 1995], DynaSee[Zayour 2000], GSEE [Favre, 2001], CodeCrawler [Lanza, 2003a, Lanza, 2003b] and Sextant [Schäfer, 2006].

In the 80s, two kinds of work appear to assist the tasks of source code understanding: **(i)** *grep*[1]-like tools, such as *grep, egrep*, and *fgrep*, which match regular expressions, and **(ii)** tools that detect plagiarism in programs [Grier, 1981, Berghel and Sallach, 1984, Madhavji, 1985].

One of the first work towards a tool for reverse engineering was proposed by Paul [Paul, 1992]. In his work, he proposed the *SCRUPLE*, *A Reengineer's Tool for Source Code Search*, which focuses on source code search, addressing the automatic detection of source code sections that fit patterns defined by the user in a pattern language.

In 1993, Müller [Müller et al., 1993] presented a new perspective for the Rigi, *a*

[1] http://www.gnu.org/software/grep/

model and tool for programming-in-the-large, which was to understand software systems using reverse engineering technology perspectives from the project [Müller and Klashinsky, 1988]. Four years later, Singer [Singer et al., 1997] performed an examination of the software engineering work practices, and discussed the advantages in considering work practices in designing tools for software engineers. Moreover, it was presented three functional and seven non functional requirements for a tool that support systems comprehension, using them to define the *Tksee* tool.

Storey et al. [Storey et al., 1999] studied cognitive design elements to support the construction of mental model during software exploration in a work published in 1999. They described a hierarchy of cognitive issues that should be considered during the design of a software exploration tool. In addition, the work described how these cognitive design elements may be applied to the design of an effective interface for software exploration and applied the framework to the design and evaluation of a tool called *SHriMP – Simple Hierarchical Multi-Perspective* tool.

In 2000, Zayour and Lethbridge [Zayour and Lethbridge, 2000] used a methodology based on cognitive analysis that is aimed towards maximizing the adoptability of a tool. They applied cognitive analysis to identify difficult aspects of maintenance work, and then derived cognitive requirements to address these difficulties. Thus, they described the approach in the context of the implementation of a reverse engineering tool called *DynaSee*.

One year later, Favre [Favre, 2001] claimed that large software products are difficult to understand because they are made of many entities of different types in concrete representations, usually not designed with software comprehension in mind. Thus, he proposed the *GSEE: a Generic Software Exploration Environment,* made of an object-oriented framework and a set of customizable tools that permit, with only a few lines of implementation, to produce graphical views from virtually any source of data.

In 2003, Lanza and Ducasse [Lanza and Ducasse, 2003] presented the concept of *polymetric view*, a lightweight software visualization technique enriched with software metrics information. The work discussed benefits and limits of several predefined polymetric views that were implemented in the *CodeCrawler* tool. In the same year, it was published a set of lessons learned in building the tool [Lanza, 2003], including the implementation.

Recently, Schäfer et al. [Schäfer et al., 2006] built the *SEXTANT Software Exploration Tool*. In this work, a set of functional requirements for software visualization and exploration tools are discussed. The paper also presents the *SEXTANT* tool based on these requirements, and discusses it with respect to the requirements of other three works that discuss comprehension support with respect to cognition, also related in our work [Singer et al., 1997, Storey et al., 1999, Zayour and Lethbridge, 2000].

Thus, based on the survey, we analyzed and grouped the requirements of reverse engineering tools, and identified a set of six functional and three non functional requirements needed to the development of an effective reverse engineering tool. Next, we discuss these requirements in details. Table 1 shows the compliance of the tools with each requirement.

FR1. Visualization of entities and relations: In 1992, Harel [Harel, 1992] claimed that "*the use of appropriate visual formalisms can have spectacular effect on* *engineers and programmers*". In general, the community has a common sense that the easy visualization of the entities of a system and these relationships, usually named and presented by a Call Graph, are important issues of a software visualization tool, mainly because a graphical presentation of entities and its relations provide an easy understanding and a useful representation of the entire system and subsystems. In this sense, Ware [Ware, 2000] claims that *other possible graphical notations for showing connectivity would be far less effective*. Thus, almost all of reverse engineering tools have a structure of call graph that allow the visualization of systems entities and relationships. The *Rigi* project focuses on this type of visualization to achieve the goals of *readability and ease understanding of system description*, and to help to *define system structure*. Moreover, the *Tksee* project shows this kind of structure to *display all relevant attributes of the items, and all relationships among them*. In addition, the *SHriMP*, *GSEE*, *Code Crawler* and *Sextant* tools provide this type of functionality.

FR2. Abstraction mechanisms and integrated comprehension: The understanding of large software systems is a hard task. The visualization of the entire system in one single view usually presents a lot of information that is difficult to understand. Thus, the capability to present several views and abstraction levels as well as to allow the user to create and manipulate these views is fundamental to the understanding of large software systems. In this sense, the *Rigi* and the *SHriMP* tools provide specific facilities to create abstractions and generate new views of the system.

FR3. User interactivity: As mentioned previously, the capability of creating user abstractions and views of a system is a desirable requirement in a software reverse engineering tool. In addition, other interactivity options are also important, such as the possibility of the user to annotate the code, abstractions and views. This type of functionality facilitates the exchange of knowledge between users, and allows the same user to remember his thoughts after some time, potentially avoiding duplicate work. Other important interactivity issue is the possibility to show to the user an easy mechanism to switch between the high level code visualization and the source code, to allow the view of the two kinds of code representation without losing cognition information. In this sense, almost all of the studied reverse engineering tools have this type of user interactivity, except for *TkSee* and *DynaSee*.

FR4. 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, and has focus on all studied reverse engineering tools, with exception of *Code Crawler* and *Sextant*.

FR5. Trace Capabilities: The software reverse engineering is a task that requires a large cognitive effort to maintain the followed paths in memory. In general, the user spends many days following the execution path of a requirement to understand it, and it is often difficult to mentally recover the execution path and the already studied items. Thus, to prevent the user from getting lost in the execution paths, the tools should provide ways to backtrack the flows of the user, show already visited items and paths, and indicate options for further exploration. In this sense, the *TkSee*, *SHriMP*, *DynaSee* and *Sextant* tools have special focus on providing tracing capabilities to the user.

FR6. 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 represented, for example, using colors, lengths and the formats of entities in a call graph. In this sense, *Rigi* tool provides some metrics in the visual presentation, and the *CodeCrawler* introduced the concept of *Polimetric Views*, that is a visual approach to enhance the visual presentation of a system. The approach consists in the possibility of *enriching the basic visualization method by rendering up to five metric measurements on a single node simultaneously,* based on node size (width and height), node color and node position (X and Y coordinates). They present examples of metrics that can be applied, which can be **(i)** module metrics: number of methods or functions extended, number of attributes and sum of LOC over all methods, among others; **(ii)** method metrics: method lines of code, number of parameters and number of method messages sent, among others; and **(iii)** attribute metrics: number of direct accesses from outside its class, number of direct accesses from within its class and number of times directly accessed.

NFR1. 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. The need for cross-artifact navigation has been identified in the context of a field study during the corrective maintenance of a large-scale software system [Mayrhauser and Vans, 1997]. In this field study, requirements on tool capabilities were derived based on developers' information needs; the most important ones concern navigation over arbitrary software artifacts. Thus, a reverse engineering tool should be capable of dealing with several kinds of artifacts, and *Rigi*, *TkSee*, *GSEE* and *Sextant* tools provide support to it.

NFR2. Extensibility: The software development area is in constant evolution. The technologies and tools are in constant change, and their lifetime is even shorter. Therefore, due to the fact of diffusion of a wide number of programming language dialects – a phenomenon known as the "500 language problem" – [Lammel and Verhoef, 2001], it is desirable that a reverse engineering tool is not able of being used with only a specific language, technology or paradigm, but should be flexible, extensible, and not technology-dependent, in order to permit its usage with a high range of systems and increasing its lifetime. In this sense, the *Rigi, TkSee* and *Sextant* are capable to be extensible to be used in reverse engineering activities of more than one technology.

NFR3. Integration with other tools: Several tools were developed to aid in reverse engineering tasks, as well to help the forward engineering. In addition, tool developers cannot foresee all contexts in which it will be used. Thus, as software reuse researchers advocate [Krueger, 1992], it is not necessary to reinvent new solutions when others already exists, and a tool should permit that features present in other tools could be incorporated in it, adopting standards to permit communication between distinct tools. In this sense, the architecture of *TkSee*, *DynaSee* and *Sextant* provides capabilities for integration with other tools.

Table 1 shows the relation between these works about reverse engineering tools and the identified requirements. In the table, the "X" indicates that the requirement is satisfied, totally or partially, by the work. A blank space means that the requirement is not even addressed by the work. By analyzing the table, it can be seen that there are some gaps in reverse engineering tools and important requirements are not considered by them, which often implement only a subset of these requirements.

In general, the tools have capabilities of entity-relationship visualizations and search capabilities, which are the base of software exploration. However, important issues such as abstraction mechanisms, metrics support and trace capabilities are present in only a small group of tools, and none of them support these three requirements at all.

Requirement	Tools							
	Scruple	Rigi	TkSEE	SHriMP	DynaSee	GSEE	Code Crawler	Sextant
Entity Relationship Visualization		X	X	X		X	X	X
Abstraction Mechanisms		X		X				
User Interactivity	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
Integration with Other Tools			X		X			X

Table 1: Relationship between the works on Reverse Engineering Tools and the requirements

In addition, most of the tools address reverse engineering with focus on architectural recovery, instead of the recovery of system requirements. Thus, we conclude that a lack of tools focused on requirements recovery, instead of pure architecture recovery, still exists. Furthermore, in agreement with flaws of reengineering and reverse engineering area presented in Section 2, and based on industrial experience of the authors and the institutions involved in this work [Fontanette et al., 2002, Alvaro et al., 2003, Almeida et al., 2004, Garcia et al., 2004, Garcia, 2005, Almeida et al., 2007, Brito et al., 2007a], we identified lacks in recovery of entire systems (interface, design and database) and to trace the requirements from user interface to database access, the difficulty of managing huge data amount, and the high dependency on the expert's knowledge. Thus, in order to fulfill these lacks, next Section presents the LIFT – Legacy InFormation retrieval Tool.

4 LIFT: Legacy Information Tool

Based on the survey that identified the main approaches in reengineering and reverse engineering areas, with their strong and weak points, presented in Section 2, as well as the study of reverse engineering tools presented in Section 3, we defined, designed, and implemented a tool for reverse engineering, focused on extracting the requirements of legacy systems, in general performed by engineers with low knowledge about the systems which many times have little or no documentation.

4.1 Requirements

In the previous section, we discussed the main requirements of eight reverse engineering tools, and identified some gaps and important requirements that are not considered by them, which in general implement only a subset of these requirements. In addition, we presented these requirements to an experienced team of the *Pitang Software Factory*[2], which had already performed reverse engineering of almost 2 million lines of code in 2006. Furthermore, the experience of the C.E.S.A.R[3] and RiSE Group[4] were considered in the definition of the requirements for the LIFT tool.

The functional requirements identified in the existent tools are: **(FR1)** visualization of entities and relations, **(FR2)** abstraction mechanisms and integrated comprehension, **(FR3)** user interactivity, **(FR4)** search capabilities, **(FR5)** trace capabilities, and **(FR6)** metrics support. The non functional requirements identified in these tools are: **(NFR1)** Cross artifacts support, **(NFR2)** Extensibility and **(NFR3)** Integration with other tools.

In addition, based on the lack of reengineering approaches discussed in Section 2, we defined two new functional requirements: **(FR7)** the recovery of the entire system (interface, design and database), and **(FR8)** the trace of requirements from interface to database access. Furthermore, in conjunction with the industry involved in this study, we defined a new functional requirement, which is **(FR9)** possibility of semi-automatic suggestions. Finally, in agreement with the literature and with the industry group, new non functional requirements **(NFR4)** scalability and **(NFR5)** Maintainability and Reusability were included in the tool. Next, we discuss these new requirements.

FR7. The recovery of the entire system: *Many reverse engineering tools concentrate on extracting the structure or architecture of a legacy system with the goal of transferring this information into the minds of the software engineers trying to understanding it* [Müller et al., 2000]. However, the software structure is not the only useful information. Most software systems for business and industry are information systems, and maintain and process vast amounts of persistent business data. Thus, the understanding of the data that is stored by the system is important. However, the research in data reverse engineering has been under-represented in the software reverse engineering scenario, and these two concepts (data and software reverse engineering) are separated. While the main focus of code reverse engineering is on

[2] Pitang Software Factory – www.pitang.com
[3] Recife Center for Advanced Studies and Systems – www.cesar.org.br
[4] The RiSE group has been involved in 6 industrial projects related to reuse since 2004 – www.rise.com.br

improving human understanding about how this information is processed, data reverse engineering tackles the question of what information is stored and how this information can be used in a different context [Müller et al., 2000].

In addition, the user interface contains the information presented to the user, and required from him, and contains lots of information about the business rules of the system.

In this sense, we believe that the possibility of recovering the entire system including user interface, the general design, and at least the database structure in a single tool should be addressed by an effective reverse engineering tool.

FR8. The trace of requirements from interface to database access: In Section 3, we discussed requirement **(F5)** Trace capabilities, which is related to the reduction of the cognitive effort of the user in reverse engineering tasks. In addition, we believe that other form of tracing is desirable. The new requirement (F7) defines that it is important to recover the entire system, from interface to databases. However, it is not important only to recover it, but also to isolate and show to the user the execution paths of application from user inputs in the interface until the persistence layer. Moreover, due to the fact that large systems contain many execution paths from interface to persistence, including loops, recursive functions and accesses to functions that do not flow to persistence, it is desirable that a reverse engineering tool provides capabilities to simplify these presentations, such as showing the minimal paths from interface to the persistence layer.

FR9. Possibility of semi-automatic suggestions: In general, the software engineer's expertise and knowledge of the domain are important in reverse engineering tasks [Sartipi et al., 2000]. However, in many cases this expertise is not available, adding a new drawback to the 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 reverse engineering tools.

NFR4. Scalability: Legacy systems tend to be large systems, containing thousands or millions of lines of code. Thus, it is necessary that reverse engineering tools that deal with legacy systems are scalable. In this sense, academy and industry both agree that scalability is one of the major issues that reverse engineering tools are confronted [Mayrhauser and Vans, 1997, Lanza and Ducasse, 2003, Schäfer et al., 2006], and that this is a requirement that must be addressed in the development of new reverse engineering tools.

NFR5. Maintainability and Reusability: As previously mentioned, LIFT project is engaged with a reuse group. Thus, software reuse researchers [McIlroy, 1968, Krueger, 1992, Heineman and Councill, 2001] advocate that the systems should be developed using software components, or other kinds of reusable artifacts, in order to provide good maintainability and reusability.

We do not believe that the identified requirements are the complete set of requirements for a reverse engineering tool. However, since they were identified after extensive studies, we believe that they are the basis for the development of an effective reverse engineering tool.

Based on these nine functional and five non functional requirements, we defined the architecture of the LIFT tool. Next, we present the architecture and implementation details.

4.2 Architecture and Implementation

Based on the requirements defined in the previous sub-section, we defined the LIFT architecture in components and modules aiming to satisfy the requirements presented in the last sub-section. The architecture defines the most important components, and expansion and integration points. The tool architecture is shown in Figure 1, and consists of four components: Parser, Analyzer, Visualizer and Understanding Environment.

4.2.1 Parser

The parser is responsible for organizing the available system data. The component is composed by two modules: The **(i)** parser and the **(ii)** pre-processing.

The parser module acts by dealing with the source code. It receives the source code, parses it and inserts all statements in a structured database. The first version of this module was developed from scratch with support of *Pitang Software Factory* team, using .NET technologies, and deals with NATURAL/ADABAS legacy source code.

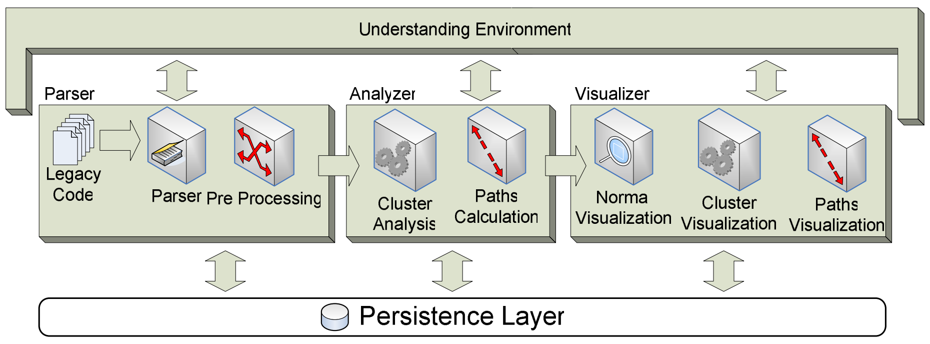


Figure 1: LIFT Architecture

The pre-processing module is responsible for receiving the parsed code and collecting the information that will be used by the tool. It accesses the parser's output, which contains all system statements, processes the information and stores it in the database structure that is used by the other tool components, focused on system modules and relations, instead of code statements. In addition, in the current version of the tool, the pre-processing is responsible for performing the program slicing, which is the identification of interface and business modules, by analyzing the modules header, which contains special characters that identify the maps (interface modules), programs and subroutines. Moreover, the pre-processing module is also responsible for the identification of the system database, by the analysis of database access statements and the identification of accessed database entities.

The separation of the parser component in two modules is useful to allow scalability, because the tool accesses the structure that contains only useful information, instead of all source code statements. In addition, this separation also allows an easy use of the tool with several technologies, since the use of a different input language can be made only by changing the parser component and adapting the pre-processor, if needed.

4.2.2 Analyzer

The analyzer component plays the role of analyzing the pre-processed code stored in the structured database and to generate representations. First, the call graph is generated containing all application modules, including and differentiating the interface and program modules, and database entities. In addition, this call graph contains other information, such as module size and the source code comments existent in the beginning of each module, which provides useful hints to the engineer performing reverse engineering tasks.

Still within the analyzer, a second step is performed, to analyze the code and deduce useful information. We defined two kinds of information to be recovered in this step, and partitioned it in two modules: **(i)** path module and **(ii)** cluster module.

The **path** module is responsible for allowing the user to follow the application paths. In this sense, it calculates the complete paths of application, and the minimal paths from interface and business modules to database modules. To build the entire paths, the module simply follows application calls, starting from interface and business modules. In addition, the minimal paths are calculated from these modules to database modules, in order to support the user in following the system sequences and to allow the tracing of requirements from interface to database access. The minimal path implementation is based on the well-known Dijkstra algorithm [Dijkstra, 1959].

The **cluster** module is responsible for identifying and showing legacy system clusters that can be recognized as a higher level abstraction, an object or component, or modules that can be merged to form one more cohesive structure. Thus, the identified clusters can be analyzed separately, and can lead to a requirement.

There are several cluster techniques, and we can highlight the K-means, Hierarchical Clustering and Density-based Clustering techniques [Tan et al., 2006].

In spite of k-means techniques being used in some reverse engineering approaches [Sartipi et al., 2000], their limitations have strong impacts in cluster detection in legacy systems. In general, **(i)** legacy clusters have different sizes and densities, with a lot of outliers; **(ii)** in most cases their modules relationships do not have a notion of a center, and **(iii)** it is user dependent. On the other hand, hierarchical clusters provide good results, but it needs expensive computational and storage requirements. Finally, density-based techniques are relatively resistant to noise that occurs in legacy systems graphs, but have trouble when clusters have widely varying densities, in addition to the expensive computational requirements.

In this context, the Hierarchical Clustering technique was chosen to perform Graph-based cluster detection in the call-graph. Thus, we chose the Mark Newman's edge betweenness clustering algorithm [Girvan and Newman, 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.

4.2.3 Visualizer

The visualizer is responsible to manage the data generated by other modules, and to present them to the user in an understandable way. Visualization is based on the call graph generated by the analyzer component and has four modules: **(i) normal visualization**, which presents the simple call hierarchy, **(ii) paths visualization**, which presents options to an easy comprehension of application paths and **(iii) cluster visualization**, which presents options to show the clusters.

The **normal visualization** presents the call graph hierarchy using the concept of *Polymetric Views* [Lanza and Ducasse, 2003] to show additional information in the graph. Thus, the visualization allows the user to configure modules and edge properties of thickness, color, format and size, according to his/her preferences. For example, the default visualization shows module colors according to the layer: blue for screen modules, green for business modules and red for database entities; and the module format varies according to the sum of its inputs and outputs. Moreover, the size of all modules is fixed by default. Furthermore, all these options are configurable. The user can change the colors, size and format of modules. Finally, the user can apply visual transformations on the graph, such as moving the modules, and performing zoom and rotation.

The **path visualization** derives from the normal visualization, and was created to provide a cleaner visualization of the paths followed by the application. In this module, the user defines how deep and to which direction (forward, upward or both) the visualization should be. Thus, when the user selects a module, only the modules in the path are shown. For example, if the user sets deep to *one* and mode to *forward*, when he selects a module only this module and all directly accessed modules are shown. With this visualization, the user can easily follow both top-down and bottom-up application paths.

The **cluster visualization** focuses on cluster detection. It allows the user to perform cluster calculations, by the choosing of numbers of edges to be removed to the graph. Thus, the clusters are calculated and repainted, the modules of the same cluster are painted with the same colors and the removed edges are painted with a weak line. In addition, to repaint, the clusters can be automatically grouped.

4.2.4 Understanding Environment

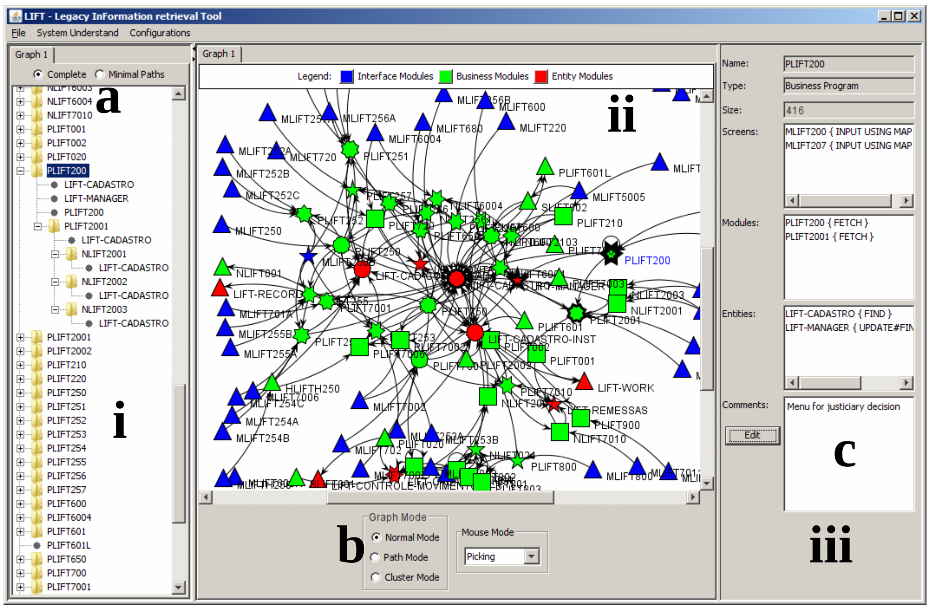
This component is responsible for integrating the other components, containing graphical interfaces for the tool functionalities.

The graphical interface for the parser component contains a single screen requiring the database connection to be used, the name of legacy system and source directory or file of legacy system. The code analysis is performed automatically, and do not have user interface. Finally, the visualizer component has several interactions with the user.

The main screen of the understanding environment has basically three areas: **(i) the path area** in the left, **(ii) the graph area** at the center, and **(iii) the details area** in the right. Figure 2 shows the main screen of the tool.

The path area contains a tree structure that shows the complete and the minimal paths of the application, and a choice button at the top (index a) providing an easy way to switch between them.

At the center, the graph area provides the interface and user interaction with the visualizations: normal, path or cluster visualizations. The switch between the visualizations is performed by choosing a choice button (index b), as well as in the switch of paths. Additionally, regardless of the type of visualization being performed, the tool allows the user to view and comment source code, maintaining both the original and the commented versions.


LIFT - Legacy InFormation retrieval Tool
File
System Understand
Configurations
Graph 1
Legend: 🟦 Interface Modules 🟩 Business Modules 🟥 Entity Modules
Graph Mode
Mouse Mode
Edit
a
b
c
i
ii
iii

[✓ Complete] Minimal Paths

Graph Mode: [✓ Normal Mode] Path Mode Cluster Mode

Mouse Mode: Picking

Name: PLIFT200

Type: Business Program

Size: 416

Screens: MLIFT200 { INPUT USING MAP
MLIFT207 { INPUT USING MAP

Modules: PLIFT200 { FETCH }
PLIFT2001 { FETCH }

Entities: LIFT-CADASTRO { FIND }
LIFT-MANAGER { UPDATE#FIN

Comments: Menu for justiciary decision

Figure 2 :LIFT main screen

Finally, the details area shows module details. This area includes the name, type and size (LOC) of a selected module. In addition, it contains its relationships, with singular areas to screens, modules and entities, and shows the relationship command, such as database accesses or module calls. At end, there is a comment area (index c) that is initially loaded with the source code comments located in the beginning of source code file, extracted in pre-processing. The comments area can be edited, in order to provide a place where the user can insert new information in addition to original code comments.

The three areas are integrated. When a user chooses a module in the path area, it is selected in the graph area and its details are shown in the details area. In the same mode, when the user selects a module in the graph area, it is selected in the path area and its details are shown in the details area.

Moreover, the tool works with the concept of code views. Thus, users can generate and deal in parallel with new subgraphs 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, and have an area that permits to the user document the view, with the insertion of its name and details.

In addition, the component has search capabilities. Since the source code is stored in database, the understanding environment uses its capabilities to perform search in the view and module comments, and in the source code.

The current version of LIFT implementation contains 76 classes, with 787 methods, divided into 25 packages, containing almost 10.000 line of code (not counting code comments). Next, we discuss the requirements compliance of the tool.

4.3 Requirements Compliance

LIFT architecture was defined to be compliant with the requirements identified in previous section. Initially, the architecture was defined aiming reusability and maintainability (Requirement NF5), being composed by independent components and modules with well defined interfaces. The main components are: (i) Parser, (ii) Analyzer, (iii) Visualizer and (iv) Understanding Environment.

The parser component is composed by two independent modules, responsible respectively for the parser and pre-processing. The parser module can be developed to deal with the target language, or an already developed parser can be attached to the tool. This capability allows an easy use of the tool with several technologies, since the use of a different input language can be made only changing the parser module. Thus, these capabilities accomplish the *extensibility* (NF2) and *integration with other tools* (NF3). In special, for a tool to be used as a parser by LIFT, it only needs a pre-processor to read its output and to store the information needed by LIFT in the database, or the tool itself can store these information, which is basically information about the modules and relations, in the database. Furthermore, the tool is not restricted to source code. The parser and pre-processor can be extended to deal with other kinds of artifacts, such as documents and domain analysis artifacts, in order to support a "sandwich" approach [Frakes et al., 1998], with both top-down and bottom-up activities, satisfying the *cross artifact support* requirement (NF1). In addition, the storage of source code information in a database system, instead of maintaining information in memory, is a fundamental item to accomplish the scalability requirement (NF4), because it permits access to source information in a dedicate server. Furthermore, the size of system does not have impact on the tool, due to the fact that database systems have special capabilities to deal with large data amounts. Finally, the access to a pre-processed data instead of the structure with all source code statements collaborates to reduce computational effort in database accesses and increasing the scalability (NF4).

The analyzer component identifies the system database structure and classifies the application modules in interface and business modules, accomplishing the requirement of *the recovery of the entire system (interface, design and database)* (F7). In addition, *the trace of requirements from interface to database access* (F8)(F5) is accomplished by the capabilities of minimal path calculations and path generations, and the cluster detection allows the *possibility of semi-automatic suggestions* (F9).

The visualizer component shows the call graph structure, allowing the *visualization of entities and relations* (F1). It also provides the possibility of system visualization and exploration. It shows the call graph with *metrics support* (F6). Furthermore, it has options to show the system in three manners: normal, path mode and cluster mode; and provides to the user options to *configure and interact* with the system in each mode (F3). Finally, the understanding environment component integrates the other components, providing the visualizations, user interactivity, creation of views (F2, F3, and F5) and *search capabilities* (F4).

4.4 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. Next, the graph is generated.

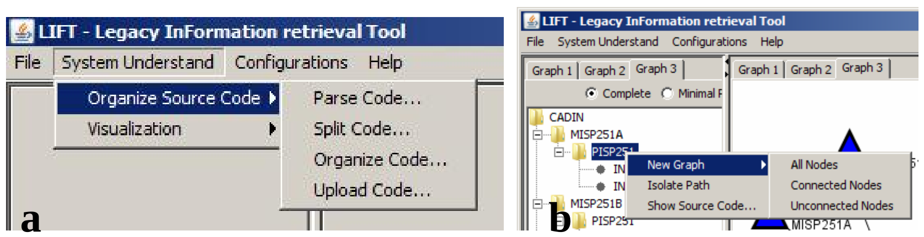


Figure 3: LIFT menus

As explained in Section 4.2 and shown in Figure 2, the main screen has three areas. The left area shows the complete paths and minimal paths from screens and business modules to database modules. In the center the call graph is shown, with the tree visualization options. The right area shows the selected module information, such as the relations and comments, inserted by the user or recognized in source code comments.

The first step to system understanding is to isolate unconnected nodes, which may be identified as dead codes or batch programs, and in general are analyzed separately from other modules. This isolation is performed by right clicking the paths area and choosing submenus "New Graph" and "Unconnected Nodes", as shown in Figure 3b.

Next, in a similar way, a new view containing only connected nodes is generated. In this view, the user tries to discover highly coupled and related modules, using cluster detection. An example of cluster detection is shown in Figure 4. 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. An example of this new view with clustered modules, and the description area are shown in Figure 5.

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

5 Evaluation

We conducted a case study in order to evaluate the effectiveness of the tool in reverse engineering projects. According to Wohlin et al. [Wohlin et al., 2000], the process can be divided into 4 main activities. The **definition** defines the experiment in terms of problem, objectives and goals. The **planning** defines the design of the experiment, the instrumentation and discusses threats to validity. The **operation** monitors the case study against the plan and collects measurements, which are analyzed and evaluated in the analysis and interpretation. Finally, the results are presented and packaged in the **presentation** and **packaging**.

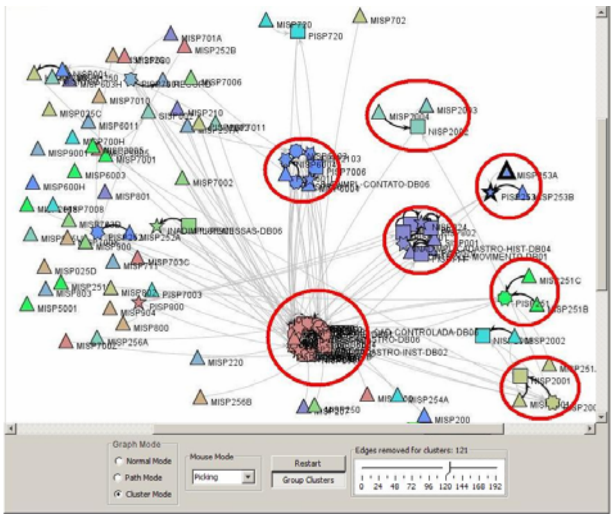


Figure 4: LIFT Cluster Detection

The case study plan presented here follows the model proposed by Wohlin et al. [Wohlin et al., 2000]. Additionally, the experiment defined by Barros [Barros, 2001] was also used as inspiration. The definition and planning activities will be described in future tense, showing the logic sequence between the planning and operation.

5.1 The Definition

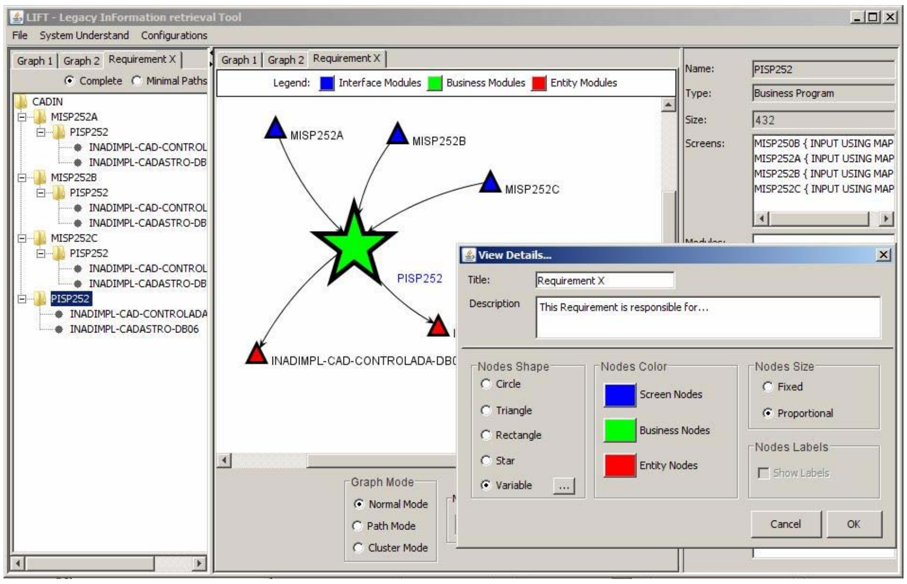
In order to define the case study, the GQM paradigm [Basili et al., 1994] was used. According to the paradigm, the main objective of this study is:

To analyze the reverse engineering tool *for the purpose of* evaluating it *with respect to* the efficiency of the tool *from the point of view of* researchers and software engineers *in the context of* software reverse engineer projects.

In addition, the questions to be answered are:

Q_1. Does the tool provide effort reduction in reverse engineering projects?

Q$_2$. Is the tool scalable to be used in large projects?



[✓ Complete] Minimal Paths

- CADIN
  - MISP252A
    - PISP252
      - INADIMPL-CAD-CONTROL
      - INADIMPL-CADASTRO-DB
  - MISP252B
    - PISP252
      - INADIMPL-CAD-CONTROL
      - INADIMPL-CADASTRO-DB
  - MISP252C
    - PISP252
      - INADIMPL-CAD-CONTROL
      - INADIMPL-CADASTRO-DB
  - PISP252
    - INADIMPL-CAD-CONTROLADA
    - INADIMPL-CADASTRO-DB06

Graph Mode: [✓ Normal Mode] Path Mode Cluster Mode

Name: PISP252
Type: Business Program
Size: 432
Screens: MISP250B { INPUT USING MAP
MISP252A { INPUT USING MAP
MISP252B { INPUT USING MAP
MISP252C { INPUT USING MAP

Title: Requirement X
Description: This Requirement is responsible for…

Nodes Shape: Circle Triangle Rectangle Star [✓ Variable] ...
Nodes Color: 🟦 Screen Nodes 🟩 Business Nodes 🟥 Entity Nodes
Nodes Size: Fixed [✓ Proportional]
Nodes Labels: Show Labels

Figure 5: LIFT view and requirement description

5.2 The Planning

In their landmark paper, Basili et al. [Basili et al., 1986] emphasize that organizations undertaking experiments should prepare an evaluation plan. This plan identifies all the issues to be addressed so that the evaluation runs smoothly, including the training requirements, the necessary measures, the data-collection procedures, and the people responsible for data collection and analysis. In addition, the evaluation should also have a budget, schedule, and staffing plan separated from those of the actual project. Finally, clear lines of authority are needed for resolving the inevitable conflicts of interest that occur when a development project is used to host an evaluation exercise.

Thus, we planned the case study as follows.

Context. The objective of this study is to evaluate the viability of using the LIFT tool in reverse engineering projects. The reverse engineering project will be conducted in a software factory in an industrial context of reverse engineering for a financial application. The software factory has experience with reverse engineering projects, with almost 2 million LOC reverse engineered in 2006. Thus, it has its proper process, staff and tools to perform reverse engineering.

Subjects. The subjects of the study will be the software factory staff. The reverse engineering activities will be performed by a system engineer. The additional roles of the process (quality engineering, configuration manager engineer, among others) will be performed by the usual staff of the organization.

Training. The training of the subjects will be conducted in meetings at the organization. The training will be divided in two steps: high level meetings, and specific training. The high level meetings will be conducted with all project staff, from managers to engineers, and will serve to show the basis and requirements of the tool, to acquire management and staff commitment, evaluate if the experiment can produce results and collect initial feedback from the project team. Three meetings will be performed, with two hour each. Next, a dedicated training will be performed with the subject that will use the tool. Three lectures will be performed with two hour each. In addition, two *use days* will happen, when the subject will use the tool with previous reverse engineered systems loaded.

Pilot Project. Due to the organization time and budget constraints, in addition to the difficulty in obtaining a small project similar to the project to be performed in the case study, a pilot project will not be performed. Instead, the subjects will use the tool with previous project data loaded, which will serve as a reference for the team.

Instrumentation. All subjects will receive a questionnaire (QT1) to gather information about his/her education and experience, with questions about strong and weak points of the tool.

Criteria. The focus of this study demands criteria that evaluate the real efficiency of the tool. The criteria will be evaluated quantitatively through the amount of effort to understand the system, related to system size in LOC, total number of modules, and the quantity of requirements recovered. In addition, the scalability of the system will be evaluated through the execution time of tasks. Moreover, the tool will be evaluated using qualitative data from questionnaire QT1. Also, all quantitative data will be compared with two other similar projects.

Null Hypothesis. This is the hypothesis that the experimenter wants to reject with as high significance as possible. In this study, the null hypothesis determines that the use of LIFT tool in reverse engineering projects does not produce benefits that justify its use and that the subjects have difficulties to use the tool. Thus, according to the selected criteria, the following hypothesis can be defined according to Wohlin et al. [Wohlin et al., 2000], as follow:

H_0': µproductivity by LOC with previous approach > µproductivity by LOC using LIFT

H_0'': µproductivity by program modules with previous approach > µproductivity by program modules using LIFT

H_0''': µproductivity by recovered requirement with previous approach > µproductivity by recovered requirement using LIFT

Alternative Hypothesis. This is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the use of LIFT tool in reverse engineering projects produces benefits that justify it use. Thus, the following hypothesis can be defined.

H_1: µproductivity by LOC with previous approach <= µproductivity by LOC using LIFT

H_2: µproductivity by program modules with previous approach <= µproductivity by program modules using LIFT

H_3: µproductivity by recovered requirement with previous approach <= µproductivity by recovered requirement using LIFT

Independent Variables. In a study, all variables in a process that are manipulated and controlled are called independent variables. The independent variables are the tool, the subject's experience, the technology, systems size and domain, the team size and the adopted process.

Dependent Variables. The dependent variables are the variables that are objects of the study which are necessary to study to see the effect of the changes in the independent variables. The dependent variables are the user productivity and tool scalability. The productivity will be measured through the effort to understand the system, related to its size, number of modules and number of requirements. The scalability will be measured through relations between system size and response times.

Qualitative Analysis. The qualitative analysis aims to evaluate the usefulness of the tool and the quality of the material used in the study. This analysis will be performed through questionnaire QT1.

Internal Validity. Considers whether the experimental design is able to support conclusions on causality or correlations [Wohlin et al., 2000]. The size of our data will be too small to allow meaningful statistical studies, so we will adopt a descriptive analysis.

External Validity. The external validity of the study measures its capability to be affected by the generalization, i.e., the capability to repeat the same study in other research groups [Wohlin et al., 2000]. In this study, a possible problem with external validity is the subjects' experience, since the subjects are experienced in reverse engineer application of the same domain and technologies of the case study. In addition, organizational factors can influence, such as the process used to perform the reverse engineer. Nevertheless, the external validity of the study is considered sufficient, since it aims to evaluate the effort reduction with the use of tool. New studies can be planned considering to use the same process in the projects to be analyzed, and subjects with similar experience, or the same subjects.

Construct Validity. Construct validity considers whether the metrics and models used in a study are a valid abstraction of the real world under study [Wohlin et al., 2000]. In this study, one of the most used legacy technology and application domain was chosen. In addition, the metrics chosen to evaluate the tool efficiency are the metrics used in real projects, such as effort in hours, and program size based on number of lines of codes, program modules and system requirements.

Conclusion Validity. This validity is concerned with the relationship between the treatment and the outcome, and determines the capability of the study to generate conclusions [Wohlin et al., 2000]. This conclusion will be drawn by the use of descriptive analysis.

5.3 The Project used in the Case Study

The project used in the case study was to perform reverse engineering of a NATURAL/ADABAS system for a financial institution. The reverse engineering was performed by one system engineer. He had just the source code of the system, without any documentation (requirements and design specification, etc). The output is the project documentation.

5.4 The Instrumentation

Selection of Subjects. For the execution of the study, system engineers of the Pitang Software factory were selected. The selection was random, where the first available engineer was selected.

Data Validation. In this study, descriptive statistics will be used to analyze the data set, since it may be used before carrying out hypothesis testing, in order to better understand the nature of the data and to identify abnormal or false data points [Wohlin et al., 2000].

Instrumentation. Before the case study can be performed, all instruments must be ready. It includes the experimental objects, the tool and the questionnaire.

5.5 The Operation

Experimental Environment. The case study was conducted during April-June 2007, at Pitang Software factory. The case study was performed directly by one engineer, and indirectly by support team (quality and test engineers, project managers, etc).

Training. The subject who used the tool was trained according to the plan.

The reverse engineering process. The subject used the habitual process of the organization which was used in the two sibling projects.

Costs. Since the subject of the case study was a software engineer of Pitang Software Factory, and the environment for execution was the organization infra-structure, the cost for the study was basically for planning and operating. The planning for the study was about three months. During this period, two versions of the planning presented in this paper were developed.

5.6 The Analysis and Interpretation

Training Analysis. The training was performed as planned. The subjects and all people involved (Pitang reverse engineer team) considered the training very good. They considered that the initial high level meetings were very important, to achieve management involvement and team motivation to use the tool, instead of traditional tools used by engineers. In addition, the subject who directly used the tool classified the dedicated training program as good and sufficient to the tool understanding. Finally, he considered that the two days were essential to clarify some questions.

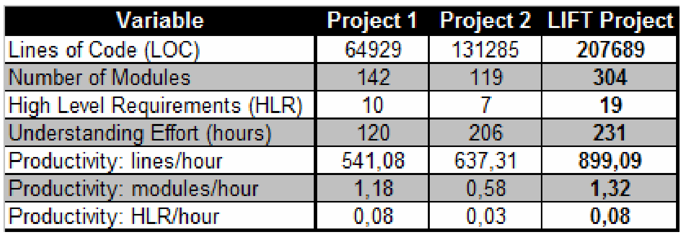
Quantitative Analysis. The analysis compares three projects, one that used LIFT tool and two other similar projects, that did not used LIFT. We call this project as *LIFT Project* and the projects that did not used the tool as *Project 1* and *Project 2*.

As explained in the Context, the three projects are similar. They use the same technology (NATURAL/ADABAS) and application domain (financial), and the subjects have almost the same experience with the technology and application domain. In addition, the projects are from the same customer, which provide similar development patterns and complexity. Furthermore, a new development team was formed, not familiarized with the specific process used to understand the systems for maintenance.

The project data was collected from two perspectives: Productivity and Scalability. The analyses were performed using descriptive statistics.

Productivity. The productivity data was obtained from the organization internal software, which the engineers use to report themselves at the beginning and end times of each activity. The other information was obtained from final system documents. Table 2 shows the comparison of productivity measures.

Lines / Hour Productivity: The engineer that performed the Project 2 was more familiar with the organization process and context. Thus, it was expected that he could produce a better productivity than the other engineers, what was confirmed in comparison with the Project 1. In addition, due to the fact that the tool introduction changes the way that users works for more than 20 years, it was expected that the LIFT project would not present much better results than the other ones. However, the productivity of *LIFT Project* was much higher than the productivity of the other projects: 66% higher than *Project 1* and 41% higher than *Project 2*. This productivity **rejects** the null hypothesis H_0', which validates the alternative hypothesis H_1: µproductivity by LOC with previous approach <= µproductivity by LOC using LIFT**.** This implies that the tool aids in effort reduction of reverse engineering tasks in system understanding, considering the size of systems in number of lines of code.



Variable	Project 1	Project 2	LIFT Project
Lines of Code (LOC)	64929	131285	**207689**
Number of Modules	142	119	**304**
High Level Requirements (HLR)	10	7	**19**
Understanding Effort (hours)	120	206	**231**
Productivity: lines/hour	541,08	637,31	**899,09**
Productivity: modules/hour	1,18	0,58	**1,32**
Productivity: HLR/hour	0,08	0,03	**0,08**

Table 2: Projects Characteristics

Modules / Hour Productivity: Even with *Project 2* having almost twice the number of lines of code than *Project 1*, the number of modules identified in *Project 2* was lower than number of modules of *Project 1*. This can indicate that in the analyzed systems, there is no relation between system module number and system lines of code. In fact, one system can have higher modularity than other, due to several causes. Despite of these differences, the *LIFT Project* presented the highest number of identified modules. Additionally, the productivity of *LIFT Project* concerning the effort by number of modules was higher than the productivity of the other projects: almost 12% higher than *Project 1* and 127% higher than *Project 2*. This productivity **rejects** the null hypothesis H_0'', which validates the alternative hypothesis H_2: µproductivity by program modules with previous approach <= µproductivity by program modules using LIFT. It emphasizes that the tool aids in effort reduction of reverse engineering tasks in system understanding, considering the size of systems in number of modules.

High Level Requirements / Hour Productivity: Although *Project 2* has almost twice the number of lines of code than *Project 1*, the number of high level requirements identified in *Project 2* was lower than the number of high level requirements of *Project 1*. As in the previous analysis, this can indicate that in the analyzed systems, there is no relation between the number of requirements and system lines of code. In fact, a requirement can need more lines of code to be implemented, or design or implementation decisions can produce different implementation of same requirement. Despite of these differences, the *LIFT Project* presented the highest number of high level requirements recovered. Additionally, the productivity of *LIFT*

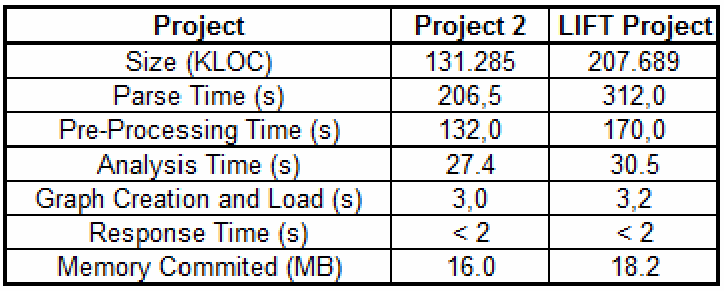
Project concerning the effort by number of high level requirements recovered was the same of *Project 1* and 167% higher than *Project 2*. This productivity rejects the null hypothesis H_0''', which validates the alternative hypothesis H_3: µproductivity by recovered requirement with previous approach <= µproductivity by recovered requirement using LIFT**.** It indicates that the tool aids in effort reduction of reverse engineering tasks in system understanding, considering the number of requirements recovered.

*Conclusion***:** Even with the analysis not being conclusive, the experimental study indicates that the tool reduces the effort in reverse engineering tasks in system understanding.

Scalability Analysis. Due to the fact that the tool was projected and implemented to be used in large systems, we studied its scalability by collecting and analyzing tasks times. In addition to the project where LIFT tool was used to perform reverse engineering, the Pitang software factory made available the source code of *Project 2,* to allow the comparison of execution times. The source code of *Project 1* could not be evaluated due to confidentiality constraints.

The scalability evaluation was performed using one PC Desktop, equipped with one Core 2 Duo processor and 2GB Ram memory, using Windows Vista operating system and SQL Server database system. This station was used both as the server and the client, in order to simulate the case that the user wants to run the entire solution in his own desktop.

In order to measure execution times, we implement a *verbose* mode in the application, which register begin and end times of each task, calculate the difference and show the execution time to the user. Thus, in the context presented, the tool performed tasks in the times shown in Table 3.



Project	Project 2	LIFT Project
Size (KLOC)	131.285	207.689
Parse Time (s)	206,5	312,0
Pre-Processing Time (s)	132,0	170,0
Analysis Time (s)	27.4	30.5
Graph Creation and Load (s)	3,0	3,2
Response Time (s)	< 2	< 2
Memory Commited (MB)	16.0	18.2

Table 3: LIFT execution times

The analysis of Table 3 indicates that the *Parse* and the *Preprocessing* code are the slower tasks, taking few minutes. However, we consider that this time does not harm the tool's performance because these tasks occur only once for each system. In addition, the *Analysis* phase takes almost a half minute, independently of the system size, but in the same way of *Parser* and *Preprocessor*, this task occurs only once by each system, so we consider this time acceptable.

In addition, execution times of user interactivity tasks are faster. The total time of *Graph Creation and Load* was about three seconds in both systems, and the total amount of heap memory committed after graph load was between 16MB and 18MB, which did not harm the overall performance of the environment (operating system, database server and LIFT application).

Moreover, we consider *Response Time* the maximum time delayed on the visualization tasks, such as to select a module to load its details, to move a module in the graph, to use the paths options and to create views options. After the analysis, we achieved that the response time of these tasks were less than two seconds in both systems, which was considered very good times by the user. On the other hand, Cluster Detection is a task that takes considerable time, in general from few milliseconds to three minutes depending on the number of clusters, modules and edges involved.

Some considerations must be addressed about these data. The initial evaluation of LIFT [Brito et al., 2007b] presented worse results than the ones described above. Based on the original results some improvements were performed in order to increase the tool performance, mainly the Graph Creation execution time. In LIFT's first version, some analyses, such as minimal paths calculation, were executed just before the graph creation every time that a system was loaded, and a lot of information was kept in volatile memory. In the new version, all analyses are performed only once, after the preprocessing phase, and all generated data is stored in the database. Thus, we reduce the time to create and to load the graph, and increase the overall performance of the environment with very little data cached on volatile memory. On the other hand, the access to these data is slower, because it is necessary to get them from the database system. However, despite of this disadvantage, the tool overall performance increased considerably.

Qualitative Analysis. After concluding the quantitative analysis of the experiment, the qualitative analysis was performed. This analysis is based on the answers defined for the questionnaire answered by the subject that used the LIFT tool.

Usefulness of the Tool. The subject reported that the tool was useful to perform the reverse engineering project. He reported that "*the tool provided some grateful help, due the fact that the documentation of existent mainframe systems is almost null, requiring a support system like LIFT to build a consistent documentation*", and that "*with the LIFT tool it became easy to generate system documentation needed to system maintenance, allowing a better visibility to legacy system*". Moreover, without having access to comparison data, he estimated that the use of LIFT reduced in almost 20% his effort in reverse engineering tasks. On the other hand, he pointed out that the main problem of the tool is the time spent to cluster calculations. This indicates that, although this time does not influence scalability, it may have some negative impact on the tool's usability. It may require some further studies to determine if additional optimizations are needed. Moreover, some improvements were discussed, such as to include in the tool the option of automatic document generation from view and module details.

Quality of the Material. The subject considered the training sufficient for using the tool. Moreover, he indicated that the presence of the experimenter in the project context was important to encourage the tool usage, due to the difficulty of changing the way he had been working in his 22 years of activities.

Additional Qualitative Analysis. Due to the fact that the experimenter was inserted in the project environment, he collected some informal user considerations about the tool.

The users agree that minimal paths visualization is very useful in knowledge recovery for re-implementation, because the main 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, he agrees 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 them.

Another important consideration is that the user reported that cluster analysis is useful to identify and isolate related modules, but the applicability of this option was limited to identify 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 points. However, cluster analysis was useful to identify some of high level requirements not included in this list, as well as clusters and sub-requirements inside them.

5.7 Lessons Learned

After concluding the experimental study, we identified some aspects that should be considered in order to repeat the experiment, since they were seen as limitations of the first execution.

Training. Although the subject claimed that the training program was good, some lectures improvements are necessary. Furthermore, some questions that still remained were clarified by the experimenter because he was allocated in the project context. Thus, in order to eliminate the need for this allocation, an online help should be included in the tool, and some kind of user support should be provided, such as e-mail contact.

Questionnaire. The questionnaire should be reviewed in order to collect more precise data related to user feedback. Moreover, a possible improvement can be to collect data about specific tool requirements.

5.8 Conclusions

Even with the reduced number of projects using the tool (one), the analysis has shown that LIFT use can help in effort reduction of reverse engineering and system understanding tasks. Moreover, it showed that the system is scalable to be used with larger software systems. Finally, the subject evaluated the tool usability as good.

In addition, the study also identified some directions for improvements. However, the study's repetition in a different context should be considered, to identify more points for improvements.

6 Concluding Remarks and Future Work

Software reengineering has been considered as a realistic and cost effective way of reusing knowledge embedded in legacy systems, instead of putting it off and rebuilding the systems from scratch. As discussed in this paper, there are several approaches and tools which perform reengineering and reverse engineering, and both academy and industry are trying new ways, such as aspect and data mining approaches. However, there are some flaws in these, in special in recovering entire system requirements, in dealing with large systems and with tools adoption.

In this sense, in order to solve the identified problems and to reduce the effort of reverse engineering activities, this work presented the LIFT – Legacy InFormation retrieval Tool. The tool is based on an extensive review of approaches and current tools, in addition to an experienced reverse engineering group's expertise. There are key differences between LIFT tool and related works. Initially, this work establishes nine main requirements that none of the related tools supported in conjunction. Moreover, we defined new requirements not attended by any of the related tools, such as cluster analysis, database induction and detection of minimal paths from interface to database modules. Finally, we defined a new way to deal with source code, which is its storage in database systems, instead of approaches that maintain a lot of data in volatile memory and harms the tool scalability.

The first version of the tool was used by an experienced organization in reverse engineering projects, with real demands of a financial institution, and presented excellent results of more than 40% of effort reduction.

Finally, a new version of the tool is being developed, with main focus on **(i) plug-ins for other input languages**, in order to allow the tool to be used in projects involving other technologies, instead of only NATURAL/ADABAS systems; **(ii) automatic documents generation**, in order to automatically generate system documentation in an automatic way, using templates defined by the user, in addition to a mechanism to trace recovered documents to the source code. Finally, we planned **(iii) more case studies** to better evaluate the tool.

References

[Almeida, E. S., 2007] "C.R.U.I.S.E. Component Reuse In Software Engineering".

[Almeida, E. S., 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.

[Alvaro, A., 2003] "Orion-RE: A Component-Based Software Reengineering Environment". 10th Working Conference on Reverse Engineering (WCRE), Victoria - British Columbia - Canada, p. 248--257.

[Barros, M. O., 2001] "Project Management based on Scenarios: A Dinamic Modeling and Simulation Approach (in Portuguese)", Ph.D. Thesis, Universidade Federal do Rio de Janeiro, p. 249.

[Basili, V. R., 1994] "The Goal Question Metric Approach". Encyclopedia of Software Engineering, p. 528-532.

[Basili, V. R., 1986] "Experimentation in Software Engineering." IEEE Transactions on Software Engineering Vol., No., p. 758-773.

[Berghel, H. L., 1984] "Measurements of program similarity in identical tasking environments." SIGPLAN notices Vol.(19), No. 8.

[Bianchi, A., 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.

[Brito, K. S., 2007a] "A Tool for Knowledge Extraction from Source Code". 21st Brazilian Symposium on Software Engineering (Tools Session), Campina Grande, Brazil, p. 93-99.

[Brito, K. S., 2007b] "LIFT: Reusing Knowledge from Legacy Systems". Brazilian Symposium on Software Components, Architectures and Reuse, Campinas, Brazil, p. 75-88.

[Canfora, G., 2007] "New Frontiers of Reverse Engineering". Future of Software Engineering (FOSE), IEEE Computer Society, p. 326--341.

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

[Connall, 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.

[El-Ramly, M., 2002] Recovering software requirements from system-user interaction traces. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering. Ischia, Italy, ACM Press**:** 447-454.

[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.

[Fontanette, V., 2002] "Component-Oriented Software Reengineering using Transformations". International Conference on Computer Science, Software Engineering, Information Technology, E-Business and Applicatinos, Foz do Iguaçu, Brazil, p. 206-211.

[Frakes, W., 1998] "DARE: Domain analysis and reuse environment". Annals of Software Engineering 5, p. 125-141.

[Garcia, V. C., 2005] "Phoenix: An Aspect Oriented Approach for Software Reengineer(in Portuguese). M.Sc Thesis." Federal University of São Carlos, São Carlos, Brazil, March/2005, p. 119.

[Garcia, V. C., 2005] "Towards an Approach for Aspect-Oriented Software Reengineering". Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, p. 274-279.

[Garcia, V. C., 2004] "Towards an Effective Approach for Reverse Engineering". Proceedings of 11th Working Conference on Reverse Engineering (WCRE), Delft, Netherlands, p. 298-299.

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

[Grier, S., 1981] "A tool that detects plagiarism in Pascal programs." SIGSCE Bulletin Vol.(13), No. 1.

[Harel, D., 1992] "Biting the silver bullet: toward a brighter future for system development." IEEE Computer Vol.(25), No. 1, p. 8-20.

[Heineman, G. T., 2001] "G. T. Heineman, W. T. Councill, Component-Based Software Engineering", Addison-Wesley.

[Jacobson, I., 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.

[Lammel, R., 2001] "Cracking the 500-language problem." IEEE Software Vol.(18), No. 6, p. 78-88.

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

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

[Lehman, M. M., 1985] "Program Evolution Processes of Software Change", London: Academic Press.

[Lientz, B. P., 1978] "Characteristics of Application Software Maintenance." Communications of the ACM Vol.(21), No. 6, p. 466 - 471.

[Madhavji, N. H., 1985] "Compare: A Collusion Detector for Pascal." T.S.I - Technique et Science Informatiques Vol.(4), No. 6, p. 489-498.

[Mayrhauser, A. v., 1997] "Program Understanding Needs during Corrective Maintenance of Large Scale Software". Proceedings of International Computer Software and Applications Conference, p. 630-637.

[McIlroy, M. D., 1968] "Mass Produced Software Components". NATO Software Engineering Conference Report, Garmisch, Germany, p. 79-85.

[Müller, H. A., 2000] "Reverse Engineering: A Roadmap". Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000). Future of Software Engineering Track, Limerick Ireland, p. 47--60.

[Müller, H. A., 1988] "Rigi: a system for programming-in-the-large". Proceedings of the 10th International Conference on Software Engineering, Singapore, IEEE Computer Society Press, p. 80-86.

[Müller, H. A., 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.

[Sartipi, K., 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., 2006] "The SEXTANT Software Exploration Tool." IEEE Transactions on Software Engineering Vol.(32), No. 9.

[Singer, J., 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.

[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., 1999] "Cognitive design elements to support the construction of a mental model during software exploration." The Journal of Systems and Software Vol.(44), No.

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

[Tan, P.-N., 2006] "Introduction to Data Mining", Addison Wesley.

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

[Ware, C., 2000] "Information Visualization", Morgan Kaufmann.

[Wohlin, C., 2000] "Experimentation in Software Engineering: An Introduction", Boston MA: Kluwer Academic Publisher.

[Zayour, I., 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.