PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 19/93

Working Results on Software Re-Engineering

Julio Cesar Sampaio do Prado Leite

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 17/93

Editor: Carlos J. P. Lucena July, 1993

Working Results on Software Re-Engineering *

Julio Cesar Sampaio do Prado Leite

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

Working Results on Software Re-Engineering

Julio Cesar Sampaio do Prado Leite
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil
e-mail:julio@inf.puc-rio.br

Abstract

We view software re-engineering is a new approach to software maintenance. Instead of performing maintenance at the source code of systems, we work on high level abstractions. From these abstractions we procceed in a forward manner reusing the available implementations, when it is the case. As such, we view re-engineering as centered on design recovery. We have been working on methods for re-enginnering and applying them to real cases. Our studies are centered on the idea of using JSD [Jackson 83] as a way of casting the recovered design. We worked with two small systems and a complex one. Our objective here is to highlight our approach, report on what has been done and point out what was learned.

1 Introduction

We understand maintenance as a broad activity embracing not only corrections on the software but also modifications needed for software evolution. Although there are authors that believe that maintenance is part of the development effort, we believe that in most cases of existing software artifacts that view can not be applied. That is, in most cases the artifacts are already dissociated from the processes that created them. As such, maintenance has to be performed independently. Re-Engineering is a well suited approach for situations where maintenance is independent of the process that created the artifact.

The idea of maintenance as a re-engineering process has been pointed by Parikh [Parikh 88] and Chikofsky [Chikofsky 90]. Several researchers have been working with a combination of reverse engineering and forward engineering to enhance the productivity of maintenance tasks. A growing community is devoting research efforts towards methods and tools that help software engineers perform maintenance [Biggerstaff 89] [Baxter 90] [Rugaber 90].

Our group at PUC-RIO[1] has been working on the idea of software re-engineering for some time now [Leite 90] [Souza 91], [Leite 91]. [Klajman 92], [Prado 92]. Work has been centered

[1]Pontifícia Universidade Católica do Rio de Janeiro

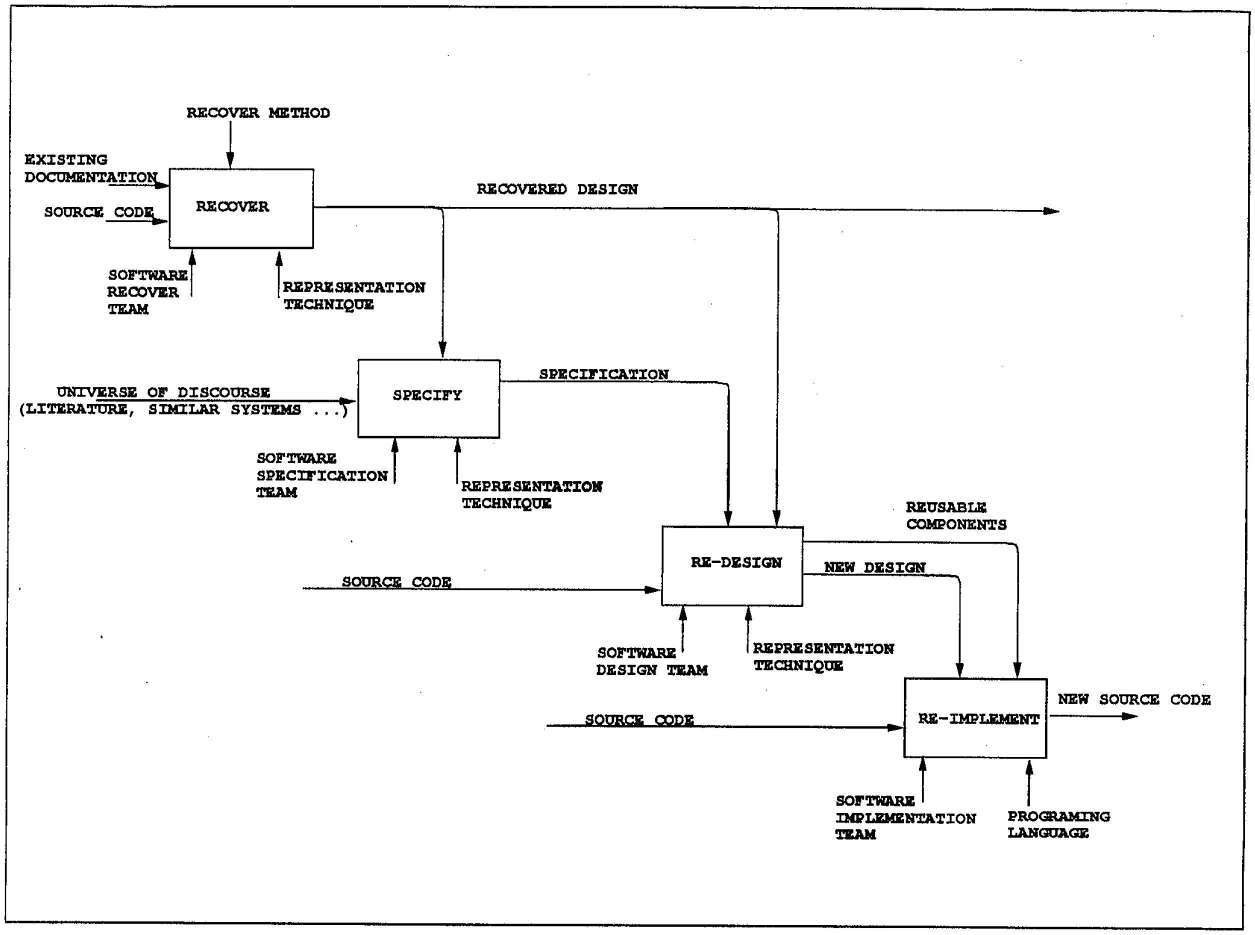

Figure 1: Re-Engineering Method

on methods and not much has been invested on tool implementation. The techniques that we developed had contributed to the effective re-engineering of two software systems and the recover of another one.

Our approach to software re-engineering is based on a meta process which considers that there are four main sub-processes to be performed when re-engineering an artifact. The sub-processes are: **recover**, **specify**, **re-design** and **re-implement**. The SADT model of Figure 1 describes the way in which those sub-processes communicated. The aim of using such a process is that it makes it possible to handle maintenance at the design level and creates opportunity for reuse, since re-implementation is guided by the previous re-design.

Recover is based on recovering designs and casting them in JSD. JSD views software development as a forward engineering activity, but we are using it backwards in order to recover designs. We have found that JSD is well fit to perform this task mainly by its uniform way of handling abstractions.

JSD is an operational specification method that uses the concept of independent and long running processes. Processes appear naturally in JSD as one encapsulates entities of the real world with its actions. Actions are real world responses to events. JSD main idea is that one has to model the problem and not the functions that a software system has to perform. Functionality appears in JSD after the problem has been modeled. We could consider the JSD method as composed of three phases: the identification of entities and actions, the specification of original and functional processes, and the implementation of processes as processors.

We have had used this meta process with different instantiations, ranging from the solely use of JSD to the combination of other methods such as Fagan's inspections and the use of Draco domain based descriptions. We will organize this report by the different experiences we have performed. Section 2 will report on the re-engineering of a hypertext system. Section 3 will report on the recover of a business system. Section 4 will report on the re-engineering of Draco, a complex software development tool. Section 5 will report on the experience in using a re-engineering approach in a large organization. We conclude, Section 6, summarizing what we have learned and pointing out what we believe should be better worked out to make it easier the task of re-engineering.

2 A Hypertext System

This case study involves a small hypertext system designed by a team of students [Maciel 90] on a software project course. The system implements a special hypertext that supports the *Language Extended Lexicon* a structure developed by us [Leite 89] to represent application vocabulary. The original system, around 5,000 lines of Pascal code (Turbo Pascal 5.5), was re-designed and re-implemented in the Unix environment in C and Sunview by a graduate student [Leite 91]. The result is HyperLex, a system that has been used in vocabulary acquisition.

The process used for this case was a specialization of the general method presented at the Introduction. It is described by the SADT model in Figure 2. Observe that, in this case, JSD is used both forward and backward. At the *recover* step JSD is used to represent the recovered design.. At the *specification* step JSD is used to represent the entities and actions of the desired new hypertext system. At the re-desing step the recovered design and the new specification are put together to see what can be reused, in terms of design, and to draw the new design.

The method used for recovering the design can be described by the following steps:

- Creation of an *Index*. The *Index* should put together the important objets perceived from code analysis. This code analysis is performed manually with the help of a cross analyzer. Below we described some of the heuristics used to create the *Index*. Observe that the heuristics are dependent on the structure of the language used.
  - each module of the system (.PAS file) will be an entry.
  - for each entry list: UNITS and INCLUDES referenced in that entry, PROCEDURES DECLARATIONS and FUNCTIONS included in that entry, and the TYPES and VARIABLES defined.
  - in the case of existing a TYPE declared as an OBJECT, list, after the type declaration, the instances variables, the PROCEDURES and the FUNCTIONS of that OBJECT.

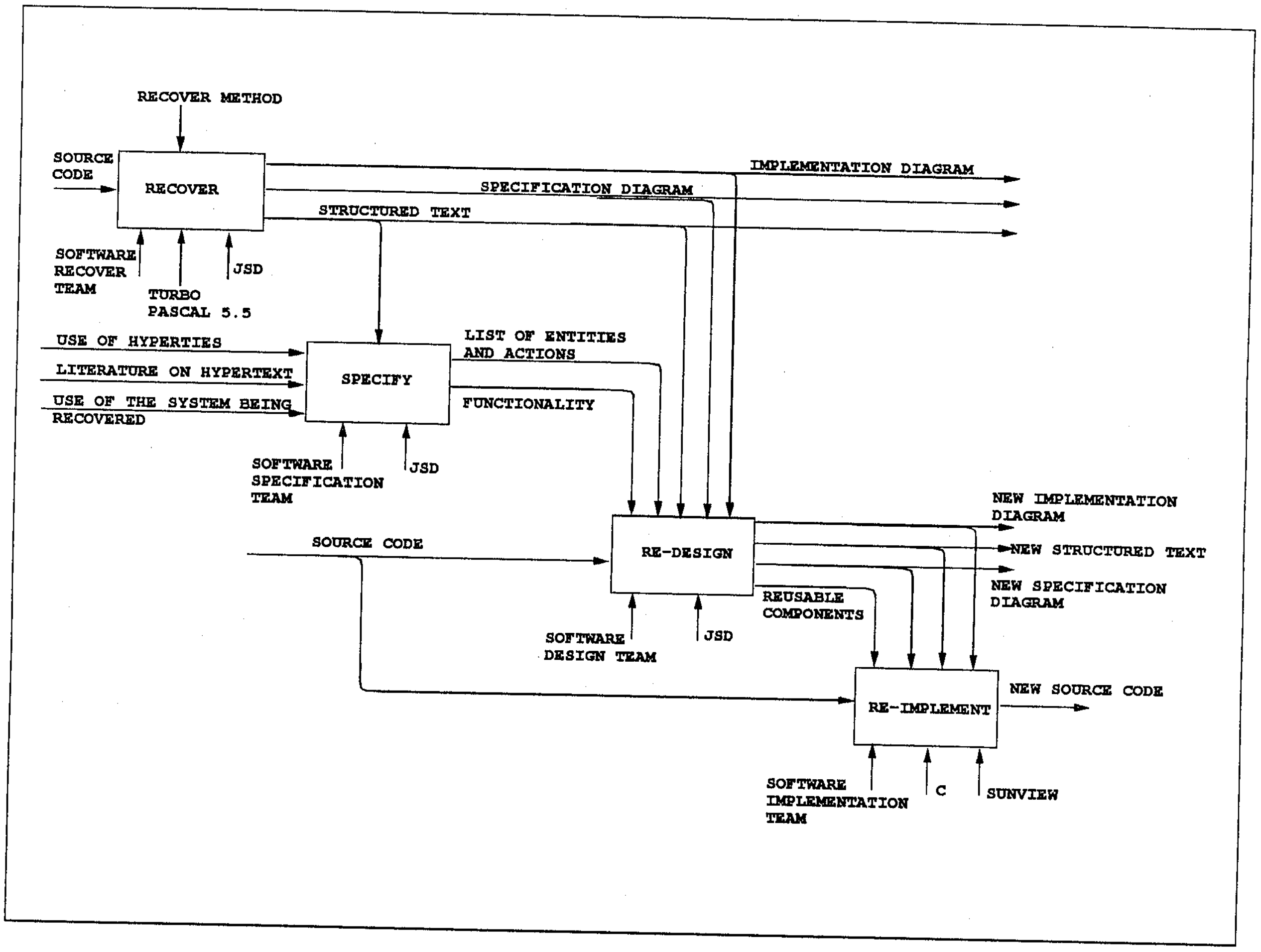

Figure 2: JSD oriented Re-engineering

- Creation of the *Implementation Diagram*. The *Implementation Diagram* is the JSD final design representation, where the process are allocated to processors. Below we list some of the heuristics used for its creation.
  - Each *object* becomes an implementation process.
  - Each *unit* becomes an implementation process.
  - The *object* in which the MAIN is declared becomes the scheduler.
  - Each file in the system becomes a *state vector* file in the diagram.
  - The system's inputs are grouped together in one *data stream* which is read by the scheduler.
  - The system's outputs are the reports or the error messages.
- Creation of the *Specification Diagram* and its *Structured Texts*. The *Specification Digram* is a network diagram of processes. In JSD, this processes are originated from the processes presented at the initial model, which pictures the problem, and from functions added later to support the system desired behavior. It is important to observe that following the guidelines, we will obtain a text that is an abstraction of the original code. Below we list some of the heuristics we have used.
  - Each process in the implementation diagram has its logic stated as an structured text using the basic components: sequence, interaction, selection.
  - Each call to a procedure or function becomes a write command to a data stream linking the calling process to the called process.
  - Acess to files are represented as state vector access.
  - The output data streams are kept as the same.
  - The inputs are divided in several data streams and attached to the relevant process.

The specification part of the re-engineering process was conducted using JSD. For each entity, found to be important, a list of actions was established. As we can see on the SADT diagram of Figure 2, the information used for the specification was drawn from several sources: the use of Hyperties, readings from the literature and the use of the system being recovered. At this step we also made a short description of the intended functionality.

Re-designing was more rewarding than we first planned. It turned out that we managed to reuse a large amount of the software. In particular we reused most of memory management and the hypertex navigation. Overall we followed the directions listed below.

- Identify in the recovered JSD specifications parts that would match the entities proposed for the new system.
- Examine the structured text of the matching parts trying to identify what could be reused.

- Identify where the interface functionality is present in both specification diagrams[2]. Cut the design such that the functions are completely separated from the original processes.
- Replace the functions with the new functions.
- Replace parts of structured text with the new requirements expressed by the list of entities and actions.
- Identify the reusable components.

In the case of HyperLex, since we were using a new target programming language, there was a need to port components written in Pascal to C. Although we had to rewrite the components, we just translated from Pascal to C, preserving as much as possible the data structures and the logic used in each module. The student who performed this task was surprised and convinced that she had saved time by re-engineering the system, instead of just re-doing everything.

3 A Business Allocation System

This case study was performed by a student as a term project. The student used as an example, the system she was working on her job at a computer manufacturer. The system, about 3000 lines of REXX code, aims at controlling the type of financial plan used by each client in buying or leasing computers.

The method applied in this case was basically the same as the one described in the case of the Hypertext. The first step was the creation of the *Index*. Using the basic components of REXX (Exec, subroutine, input file, output file), we decided that each Exec would be an *Index* entry and for each Exec there would be a list of:

- the Execs called,
- the subroutines called,
- the files used,
- the import and export variables.

The next step was the creation of the JSD *implementation diagram*. We have assumed that the code itself was the structured code, so we draw the *implementation diagram* using the *Index* and the following assumptions:

- each Exec becomes a *process*,
- the main Exec becomes the *scheduler*,
- the files are the *state vectors* files in the *implementation diagram*.

[2]We have to understand that in JSD there are two ways of handling functions: functions that are itself processes and implicit functions. Implicit functions are embedded in a process structured text at the time of designing the JSD specification diagram.

- the *scheduler* handles the interface and one *data stream* handles all the inputs.

Using the *Index* and the *implementation digram*, it was possible to write down the *structured text* for each process. This description was an abstraction of the original code, and a step towards the *specification diagram*, the final representation of the recovered design.

Regarding validation; we read the *specification digram* and believed it was coherent with the general idea of the system. In the context of the term project, the student also performed an informal validation with the original designer of the business allocation system. After explaining to the person the idea and the details of JSD, the student asked if the *specification diagram* was realistic. The designer, who hadn't work on that system for a long time, agreed that, overall, the recovered design was realistic. The student herself believes that the exercise helped her a lot in performing her tasks of maintaining the system, although it is not clear if she will continue to use the JSD method or not.

4 Draco-PUC

This case study is more complex and a more elaborated work [Prado 92]. Prado proposed an instantiation of the re-engineering method shown in Figure 1 in such a way that it could use the domain structure proposed by Neighbors [Neighbors 80] to re-design software. Neighbors, on his work on Draco, puts forward the idea that it is possible to produce software by reusing high level abstractions implemented as component libraries. The strategy proposed by Prado [Prado 92], Figure 3, is composed of four activities: **recover, specify, re-design, re-implement.** Departing from the source code, the method combines inspections [Fagan 90] and JSD to recover the design. Using the recovered design, the observations made to it and performing a more detailed study of the problem area, one can specify the desired changes. In the next step those changes are used to re-design the artifact, representing this new design as a Draco domain. Finally the design is re-implemented in a executable language, that can be different from the original implementation language.

The Draco machine can be viewed as a application generator generator, that is a meta generator. As such it has two distinguished parts, one that build the generators, called domains, and other that uses the domains to build software systems. In order to build a domain, it is necessary to build:

- a parser,
- a prettyprinter,
- a component library and
- a transformation library.

Once a domain (generator) has the parts listed above built, it is possible to construct software systems in that domain. Four main subsystems are responsible for the software construction process. Following we describe them.

- Parse: the subsystem responsible for domain programs analysis and the creation of the Draco abstract syntax tree (DAST). DAST is the basic Draco representation.

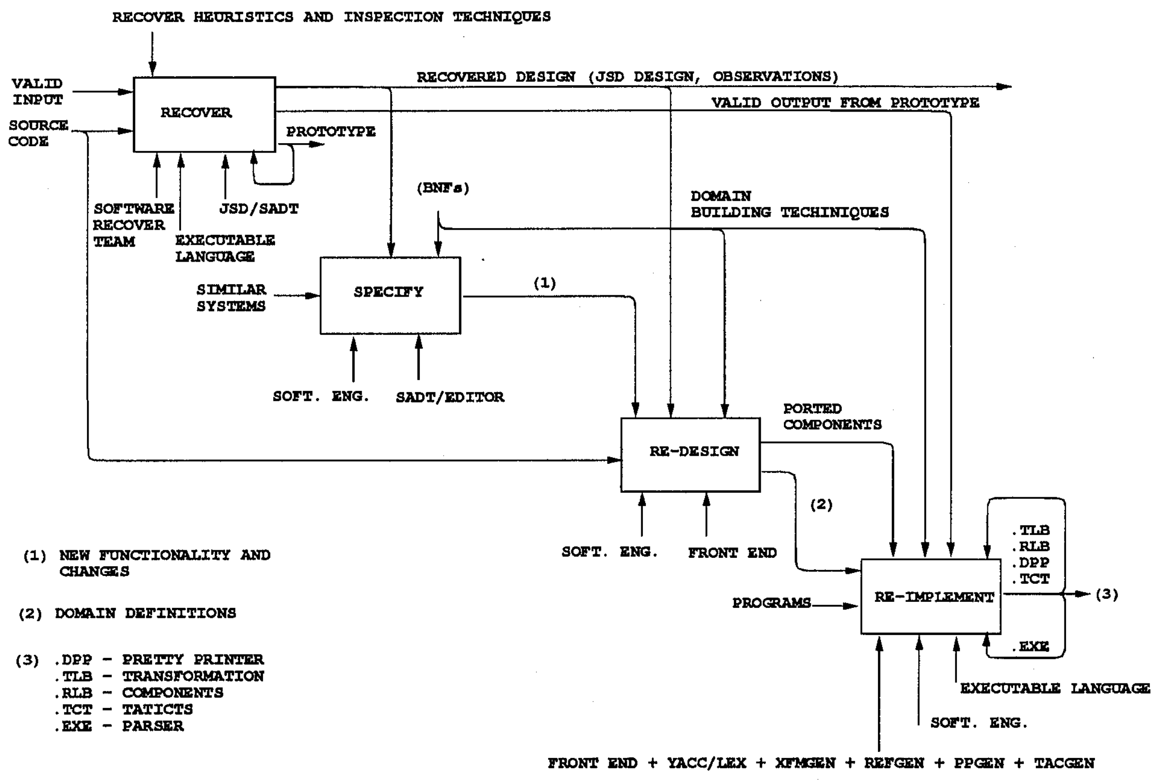

Figure 3: Prado's Re-Engineering Strategy

- Prettyprinter: the subsystem that displays the contents of the DAST using the original syntax of a given domain.

- Transform: this subsystem applies the transformation rules performing manipulations on the DAST. Those transformations are horizontal, that is they are intra domain.

- Refinement: with this subsystem it is possible to perform vertical transformations of the DAST. These transformations are inter domains. Guiding these refinements we have a set of tactics that help automate the process of translating one domain description into other domains.

Although Draco has been discussed and evaluated in different occasions [Neighbors 84], [Arango 86], [Freeman 87] [Arango 88] [Neighbors 91], the Draco prototype itself has been basically the same as the one built in 1980. Most of the work around Draco has been on the ideas surrounding it, and not on empirical work of trying to use it. Our research strategy is centered on the hypothesis that the Draco idea is sound. As such, our research agenda is built around the Draco machine.

Pursuing our goal of having a usable Draco, we have re-engineered Draco into what we call Draco-PUC. Draco-PUC version 0.1 was the result of a design recovery process [Leite 91]. It is basically a port from the original UCI-Lisp code, except for a new interface. Version 0.1 was written in Scheme. Draco-PUC version 1.0 has several re-designed parts. The major re-design was in terms of the structure of the parsing mechanism. Draco-PUC v.1.0 uses an off-the-shelf parser generator, Yacc, and has an Draco-Yacc editor that helps domain construction.

The recovered procedure used by Prado is similar to the one described in the previous examples, but with a substantial difference. In this case, an inspection process [Fagan 90] is used to help code understanding and to validate the design being recovered. The inspection process has the following steps.

- Preparation: the moderator describes the overall area and the main and intermediate goals to be achieved. The participants read the code, plus any other extra material, and the designer is responsible for representing the recovered design using JSD structure diagrams. The level of abstraction chosen for casting the design depends not only on the implementation language, but on the problem itself.

- Inspection: the designer describes each recovered structure diagrams and the implementor and the moderator ask questions. These questions are based on existing checklists and in some recover heuristics. The questioning process tries to discover any existing mistakes, errors or problems.

- Correction: the problems are fixed and a new version of the diagrams are produced.

- Validation: the moderator makes sure the new version is correct. If there is no agreement, then the whole process is repeated.

Once the design of each of the subparts are recovered, it is time to integrate each of these subparts in a JSD system specification diagram, the network model used in JSD. In this representation, the processes (subparts) interface between each other by means of data stream or state vector. Arguments are usually seen as data stream and global variables are transformed in a fictitious process, globalvar, from where access can be made by state vector or data stream.

In the activity specify (see Figure 3) changes and improvements are represented in SADT and in text descriptions. Analysis of the recovered design together with the study of similar systems, or the requirements for changes, are the main sources to the specification process. Controlling this activity we have the Draco domain representations.

In order to represent the new design, we use the Draco domain's four parts: the parser, the set of transformations, the components and the prettyprinter. As pictured in Figure 3, the specification of changes, the recovered design and the techniques for Draco domain construction are determining the way the re-design is performed. There are 5 big steps in the process of re-design:

1. grammar definition
2. components and tactics definition
3. porting
4. transformation definition
5. prettyprinter definition

Besides defining a domain for the software being re-engineered, it is necessary to have defined and implemented the target domain, here understood as an executable domain. In order to have a target domain, for instance C or Pascal, we need a parser for that domain, and a prettyprinter. A transformation library, although recommended, is not mandatory.

It is interesting to observe that by using the idea of Draco domains to the realm of programming languages, we can port systems without understanding its semantics. That is, we can write grammars for two executable languages and express the language semantics by components written in the other language. An exercise like this was executed by porting Draco itself [Arango 88].

In order to **re-implement** we need to build a Draco domain. As such we have to make operational, the parser, the prettyprinter, the component library and the transformation library of the chosen domain. The Draco-PUC machine has an editor that helps the construction of the domain parts. Once the domain is available the software engineer can use it as a generator, and as such improves the possibility of different specifications for that artifact, but reusing the same implementations of the original artifact. One needs to specify a set of programs to represent the new artifact (see Figure 3). These programs would be read by the machine and will produce the needed software in the target language of the implementor's choice.

Draco re-engineering followed the method described here. Several modifications were performed, altering the design of Draco. The subsystems for building the transformations and the tactics were written as domains and as such their coupling with the whole system was made weaker. The subsystem for building the component library was split in two: one for creating the library and another for filling the library. The re-design of the parsing mechanism used in Draco was its central change. By being able to cut the design, that is separate the parts in the recovered design, we managed to replace parts of the machine by other parts and to maintain old parts running with those new parts. The substitution of the Draco parsing mechanism and the Draco interface was a clear sign of the success of our re-engineering strategy.

Draco-PUC is a hybrid system. It has parts in C, parts in Scheme, parts in Assembler and uses the Yacc system. Its interface is similar to Borland's compilers interfaces and has a Yacc oriented editor that helps the creation of grammars for Draco domains. Draco-PUC is about 20 K lines of code (without Yacc), much different from the original Draco 4 k lines of Lisp. Much of the lines comes from the new interface.

5 Organizing Maintenance in a Large Organization

Another related experience was the proposal of a maintenance process for a very large organization. This organization is a large mining company, and has a large investment on software for corporate information systems. A member of the company's technical staff developed his master's thesis [Souza 91] within our group in order to describe a process to organize maintenance arounding the idea of re-engineering.

In Figure 4, a datagram [Ross 77] shows the main entities involved in such a process and the actions that create or use them. There is the need for the company to settle on some standards, for instance on requesting changes or to the organization of the desing library. Of course there is the need for a plan on how and when to tackle users requests. The use of the design library is such that it is in accordance with the configurator control that keeps an eye on what is changing and why. Centering the re-implementation on using the desing library, has made it clear to the company that it is important to have a *good* design library and second that it is worth using the library.

The company is implementing the process and finding out that it can be done, but has encountered the usual roadblocks: lack of support from upper management, pressure for quick fixes, and lack of supporting tools.

6 Conclusion

We have reported on ongoing research on software re-engineering. Our work has been oriented towards experimental case studies in order to stablish processes and methods to perform software re-engineering. Our process, which divides the re-engineering task in four sub tasks, **recover**, **specify**, **re-design** and **re-implement** has been applied in three different cases with *good* results. The methods used have been based on the use of JSD as a way of representing the recovered design.

Jackson in his justification of JSD pointed out that modeling the real world would make it easier to implement changes, if those changes were at the functionality level, for instance, a new report or a different combination of data in the system. As such JSD has means of differentiating between aspects pertaining to the problem and aspects related to the functionality of a given system. Although it is very hard to make the difference as one recover the design, it is true that once it becomes clear what is related to functions and what is related to the real problem, JSD representations easily handle the difference.

We have noticed, on the two small examples, that as pointed out by Jackson, most of the maintenace relates to functions. In both cases the interface was easily identified, and in the

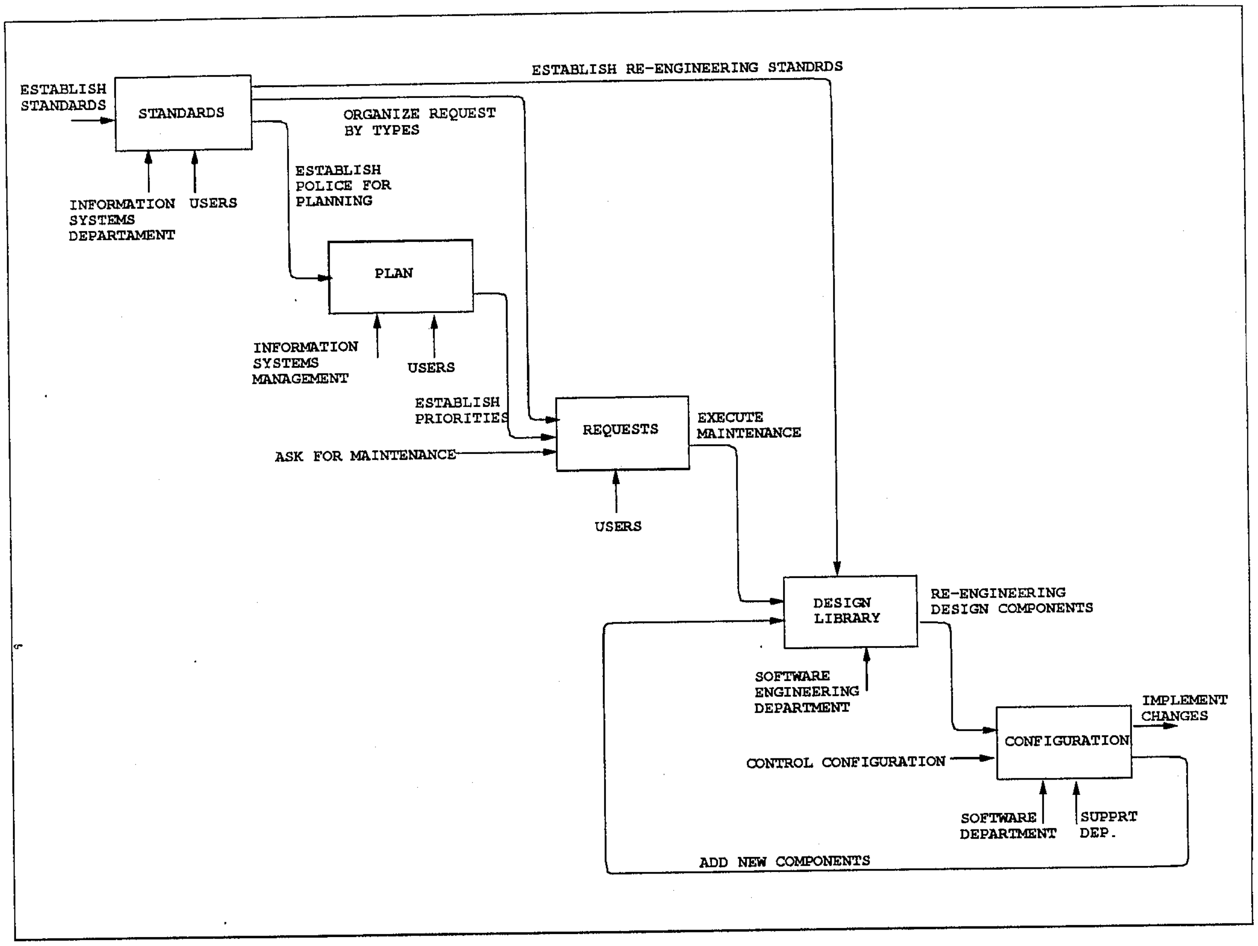

Figure 4: Organizing Maintenace as a Re-Engineering Activity

case of the hypertext system completely replaced by a complete new interface. We also noted this in the Draco re-engineering. In the case of the hypertext it was also the case that some of the reports were changed and others added, again changes performed on the functionality. With Draco the identification and substitution of the interface was made simpler by the use of JSD.

Using Draco domains for representing re-designs has been showed to be much more complex. It is not easy to deal with representations that requires a lot of detail and deep knowledge of grammar writing, which is not trivial. Nonetheless, we believed tha much can be done, an example is Draco-PUC interface, to make this task easier. We have to keep in mind that as re-designing system as a means of Draco domains we will have the additional benefit of possible reuse, being this reuse already in the Draco format.

We observe that using the inspection approach to the taks of recovering is an effective way of guarantying quality and improving the understanding of the system. We will pursue on this idea, trying ot apply it for other complex cases.

As stated in the Introduction our work is centered on method support for the manual labor of software engineers. We should however start to look for possible ways, besides cross-referencing, to automate parts of our process. Although it is debatable if our JSD based proposal is better for instance than one that would use object oriented representations, we believe that we should keep doing experiments since this case studies will provide us with more information for designing usefül re-engineering automation procedures.

References

[Arango 86] Arango G., Baxter I., Freeman P., Pidgeon C., *A Transformation-Based Paradigm of Software Maintenance*, IEEE Software, Vol. 3, pp. 27-39, May 1986.

[Arango 88] Arango G., *Evaluation of a Reuse-based Software Construction Technology*, Proc. Second IEE/BCS Conference on Software Engineering 88. The British Computer Society, July 1988.

[Baxter 90] Baxter, I. *Transformational Maintenance by Reuse of Design Histories.* PhD. Dissertation, University of California, Irvine, USA; Nov., 1990.

[Biggerstaff 89] Biggerstaff, T. *Design Recovery for Maintenance and Reuse*, IEEE Computer, 22(7), pp. 36-49, Jul. 1989.

[Chikofsky 90] Chikofsky, E. e Cross II, J. *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, pp. 222-240, Jan. 1990.

[Fagan 76] Fagan, M., *Design and Code Inspections to Reduce Errors in Program Development* , IEEE Software, pp. 222-240, Jan. 1990.

[Freeman 87] Freeman, P. Software Reusability, IEEE - Computer Society, March 1987.

[Jackson 83] Jackson, M. *System Development*, Prentice-Hall International; 1983.

[Leite 89] Leite, J.S.P., Elicitation of Application Languages. *In Monografias em Ciência da Computação*, PUC-RIo, no. 30, 1989.

[Leite 91] Leite, J.C.S.P. e Prado, A.F. *Design Recovery - A Multi-Paradigm Approach*, First International Workshop on Software Reusability, Dortmund, Germany; Jul., 1991.

[Leite 91] Leite, J.S.P. Franco, A.P.M., Re-Engenharia de Software, um Estudo de Caso, *In V Simpósio Brasileiro de Engenharia de Software*, Ouro Preto, Out. 1991.

[Parikh 88] Parikh, G. *Technics of Program and System Maintenance* (2a. edição), QED Information Sciences, Inc., 1988.

[Maciel 90] Maciel, G., Costa, J. e Baccar, J. Editor de Léxicos, *Trabalho de Fim de Curso de PSS*, Departamento de Informática, PUC-RIO, Mar. 1991.

[Neighbors 80] Neighbors J., *Software Construction Using Components*, PhD. Dissertation, Dept. Of Information and Computer Science, University of California, Irvine, 1980.

[Neighbors 84] Neighbors J., *The Draco Approach to Constructing Software from Reusable Components*, IEEE Trans. on Software Engineering, SE-10:564-573, September 1984.

[Neighbors 91] Neighbors J., The Evolution from Software Components to Domain Analysis, *V Simpósio Brasileiro de Engenharia de Software*, Ouro Preto, MG, Out. 1991.

[Prado 92] Prado, A.F., Estratégia de Re-Engenharia de Software Orientada a Dominios, Tese de Doutorado, *Departamento de Informática, PUC/RIO*, 1992.

[Ross 77] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. *In Tutorial on Design Techniques, Freeman and Wasserman (ed.)* IEEE Catalog No. EHQ 161-0(1980), 107-125.

[Rugaber 90] Rugaber, S., Ornburn, S. e Le Blanc Jr., R. Recognizing Design Decisions in Programs. *In IEEE Software*, 7(1), Jan., 1990.

[Souza 91] Souza, A. Manutenção de software sob Enfoque de Re-Engenharia, *Dissertação de Mestrado*, Departamento de Informática, PUC-RIO, Mar. 1991.