Submitted to:
ERCIM Database Research Group Workshop
EDRG Workshop 4
"Repositories, methods and tools for systems engineering"
May 3-5, 1993
ICS-FORTH, Crete, Greece

*A Repository Based Tool for Re-Engineering towards an Object Oriented Environment

Oreste Signore - Mario Loffredo

SEAL (Software Engineering & Applications Laboratory)
CNUCE - Institute of CNR - via S. Maria, 36 - 56126 Pisa (Italy)
tel. +39 (50) 593201 - Fax. +39 (50) 904052
e.mail: oreste@vm.cnuce.cnr.it

Abstract

Software re-engineering and object orientation are two areas of growing interest in the last years. However, while many researchers have focused their interest in the object-oriented design methodologies, a little attention has been paid to the re-engineering towards an object-oriented environment.
In this paper we examine the motivations towards object-oriented re-engineering (extendibility, robustness and reusability of the code) and the problems found in moving from a process-based to an object oriented perspective.
Finally, we describe the architecture of TROOP, a tool that implements the object oriented re-engineering, combining into a single repository information describing both the conventional and the object-oriented target environment.

Keywords: Repositories, Software re-engineering, Object-orientation, Program slicing

* This work has been partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R.

[*]A Repository Based Tool for Re-Engineering towards an Object Oriented Environment

Oreste Signore - Mario Loffredo

SEAL (Software Engineering & Applications Laboratory)
CNUCE - Institute of CNR - via S. Maria, 36 - 56126 Pisa (Italy)
tel. +39 (50) 593201 - Fax. +39 (50) 904052
e.mail: oreste@vm.cnuce.cnr.it

Abstract

Software re-engineering and object orientation are two areas of growing interest in the last years. However, while many researchers have focused their interest in the object-oriented design methodologies, a little attention has been paid to the re-engineering towards an object-oriented environment.
In this paper we examine the motivations towards object-oriented re-engineering (extendibility, robustness and reusability of the code) and the problems found in moving from a process-based to an object oriented perspective.
Finally, we describe the architecture of TROOP, a tool that implements the object oriented re-engineering, combining into a single repository information describing both the conventional and the object-oriented target environment.

1 - Introduction

Software engineering is evolving towards the implementation of tools that could improve the maintainability of the existing software applications, possibly by their reconfiguration.
Up to now, the research activities about the possibility of re-implementing a software system have led to the definition of two fundamental versions of the so called *re-engineering cycle*: the reverse re-engineering and reuse re-engineering.
The first paradigm allows the transformation of the existing application, either because a different language or a different data management system are to be used, or

[*] This work has been partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R.

because an interface towards a different tool must be implemented, and involves essentially two subsequent phases ([Chikofsky90]):

- a first phase in which we operate a reverse engineering (RE) action in order to identify, by also considering information from other sources, the system components of the product we have to reconfigure and their interrelationships and describe them in a new form or at a higher level of abstraction;
- a second one to execute a forward engineering (FE) step, which is in charge of implementing the new product.

The activities concerned with the second approach are claimed to give a solution to the twofold requirement of implementing more extensible and maintainable software systems and rescue existing software patrimony.

The proposals for a reuse re-engineering paradigm agree upon, firstly, identifying the components to be extracted according to some selection criteria, secondly, extracting and qualifying the candidate components and, finally, retrieving and fitting the components which totally or partially satisfy the user requirements.

Moreover, the new coming, growing and spreading out of object oriented methodologies seem to offer a completely new research area to the discipline of re-engineering.

In fact, the object oriented approach claims to lead to the design and implementation of software systems that are more maintainable and documented than the systems developed by making use of the more traditional methodologies.

As it has already been pointed out in [Meyer88], the object-oriented style makes possible to design software that fulfills several desirable code requirements:

- *Modularity*, by offering a natural support to the decomposition of the entire system into modules (*classes*).
- *Extendibility*, because the inheritance relationship makes easier the reuse of existing definitions and facilitates the development of new ones, and the type polymorphism enables to add new specialisation, without forcing modifications of the entire application.
- *Integrability*, because by means of the encapsulation mechanism the classes can integrate themselves with each other. All the interactions will take place only via a well defined interface, and hide all the implementation details to the rest of the system.
- *Robustness*, as the operators can access and modify uniquely the data pertaining to their class and interfaces act as a filter of the interaction between the classes. As a consequence, we have a reduced number of connections between the various classes.

- *Reusability*, because whenever we declare an instance of a class, we reuse data structures and operators acting on them. In addition, the inheritance supplies at high level the modelling of generalisation and specialisation relationships, at low level the reuse of an existing class as a basis for the definition of a new one.

Therefore, we may envisage the birth of a new research line in the software re-engineering area, the **OORE** (Object-Oriented Re-Engineering), whose aim is to develop theories and models supporting the re-engineering of existing applications towards programming environments like C++ or Eiffel.

As a conclusion, we can identify three different kinds of re-engineering, depending on their target ([Signore92a]):

- ***reverse re-engineering***: the target is the system itself, that will be re-documented or re-designed, possibly producing a final version implemented in a different imperative language;
- ***reuse re-engineering***: the target is a new system, re-designed reusing knowledge and design elements taken from the previous products, but maintaining the top-down design style;
- ***object-oriented re-engineering***: the target is the same application system, however designed according to the object oriented methodology.

In each one of these three software engineering disciplines, the presence of a *repository* containing heterogeneous information related to the system analysed is a fundamental issue.

In fact, for the first discipline, is well understood the importance of the availability of a repository that will contain all the information related to the application software, belonging to any detail level (from the analysis phase to the code), to any type of sources (formal or informal), to any form (graphical or textual), to both the data or processes architectures. However, it can be difficult to recover the existing software portfolio making a one-pass transformation from the poorly documented, but sometimes powerful "handicraft software" to a well documented, formal, repository based software factory environment: some information can be missing or incomplete, and can be collected only after a complete walk-through of the existing software.

These considerations lead directly to the identification of an *intermediate level repository*, where the low level details taken from the existing programs can be stored together with some "tentative" high level items, like entities and their associations, or the business functions. The information collected in this repository can be afterwards transferred into the consolidated repositories.

As far as the reuse re-engineering paradigm is concerned, the repository is useful in each stage ([Basili90], [Cimitile91]) In fact, during the *identification* phase, the repository is accessed to get the modules submitted to analysis and retrieve the metrics which can be helpful for selecting the potential components.
During the *qualification* phase, the components are given an abstraction, a functional specification and an informal description and are stored in the repository according to certain classification.
Finally, during the *selection* phase, the repository is accessed in order to obtain the components the totally or partially match the user conditions and, if a partial matching happens, adapt the retrieved components to the user application.
About the last paradigm, even if all the programmers will adopt the object oriented programming style, a complete reusability will not result.
In fact, it would be necessary for the potential users to have information about the available components.
A primary attempt of a components database is constituted by the Smalltalk browser through which the user can exploit the class hierarchy up to the code of each class.
The class repository should comprise a mechanism for the retrieval of class information to make possible queries on the class name, the code, the class features, the keywords assigned by the user and, at the same time, this mechanism should also enable the navigation in the class space by walking along the parent-child and the client-server relationships.
To summarise the concepts discussed before, the content of the repository associated to the three disciplines are presented in the following table:

DISCIPLINE	**REPOSITORY CONTENT**
Reverse Re-engineering	Diagrams, annotations, code, etc.
Reuse Re-engineering	Potential reusable components, abstractions, election criteria, metrics.
Object-Oriented Re-engineering	Classes, attributes and methods.

2 - Related work

In spite of its novelty, some relevant work has been yet done by several authors, and we can distinguish two main approaches in the area of the Object Oriented Re-engineering.

In their paper, Jacobson and Lindström ([Jacobson91]) suggest that an object oriented development method can be used to gradually modernise an old system via a three steps process. The first step consists of a reverse engineering phase, which allows to identify how the components of the system relate to each other and then create a more abstract description of the system. In the second step, reasoning about the changes in functionalities is done at a more abstract level. Finally, in the third step, a forward engineering phase takes place, redesigning the system from the abstract representation to the concrete one. In the whole process, the informal documentation (manuals, requirements specifications, etc.) is taken into account in order to reconstruct the knowledge about the system functionalities.

In this approach, it is supposed that it will be possible to migrate from a top-down design environment to an object-oriented one. This implies a hybrid Software Life Cycle model (Edwards90]), where we can map the Data Flow analysis models into Object Oriented Design techniques ([Alabiso88]).

Liu and Wilde ([Liu90]) concentrate their attention on the methodologies to aid in the design recovery of object-like features of a program written in a non object oriented language. Two complementary methods are proposed, based on an analysis of global data or of data types.

As far as the approach proposed by Jacobson and Lindström is concerned, we notice that, even if in principle the informal documentation may be of relevant importance, in practice it might happen that it is lacking or incomplete or inconsistent and misleading. Therefore, information kept from the informal documentation should be carefully examined and validated.

Liu and Wilde themselves in their paper raise the question if their approach may produce “too big” objects. This is due to the intrinsic characteristics of the proposed methods. In fact they consider as strongly connected procedures and data structures if they are used together, and in this case they identify the set of the data structures as an object and the procedures as methods of this object.

Because of this, we completely agree with the authors about the fact that “a further stage of refinement will be necessary in which human intervention or heuristically guided search procedures improve the candidate objects”.

Repositories and software classification schemes have been largely investigated by several authors in various software engineering research areas.

The *design recovery* constitutes the main issue in the Desire prototype presented in [Biggerstaff89]. The basic consideration is that source code does not contain much of the original design information, that therefore must be extracted from additional information sources, both automated and human. A great emphasis is put on the fact that the design recovery is essentially a human task, and cannot be implemented in a fully automated way. The first step to take is to identify the "modules" (not always clearly identified by appropriate language constructs), the "software engineering artifacts", the other informal design abstractions and their relationships to the code.
A second step will populate the reuse and recovery libraries. In a third step, the model-aided design recovery process takes place. It must be stressed that a great attention is paid to the use of *informal information*, that only a human intervention can understand and analyse.
A reusable object retrieval system is the main issue presented in [Sedes92]. The focus is on the retrieval stage on a hypertext containing heterogeneous documents, that are classified according to a classification model based on a combination of notion of faceted ([Prieto-Diaz91]) and hierarchical classification. The usage of an associative thesaurus and the assignment of a weight to each indexing term allow the identification of similarities between different nodes and the arrangement of the retrieved documents according to their relevance to the query.
[Basili92] focuses on software reuse and utilisation of the life cycle products from previous developments. In their paper the authors present the concept of the component factory and define a reference architecture from which specific architectures can be derived by instantiation. A significant aspect is the repository, where heterogeneous reusable products and reusable experience are stored and made accessible. A reusable software component is a collection made of a software component packaged with everything that is necessary to reuse and maintain it in the future.
The tool presented in [Burton87], named RSL, has as foundation a repository which stores the attributes of every reusable software component. Several types of software components can be entered into the RSL, including functions, procedures, packages, and programs. These components can be written in any language, even if the authors emphasise ADA over other languages. The RSL's software classification strategy is based upon the combination of two alternate mechanisms. The first one is the assignment of hierarchical category code to every component. The category code specifies the type of the component and its relationship to other components. The second one makes use of descriptive keywords not associated with the category codes, so allowing for overlapping topics.

[Helm91] combines the Information Retrieval and the domain specific approaches, to retrieve classes in an object-oriented library. This choice leads to the implementation of a class retrieval tool based on natural language queries and new kinds of browsing tools based on class functionality rather than inheritance.

3 - The TROOP tool

TROOP (**T**ool for **R**eengineering towards **O**bject-**O**riented **P**aradigm) is a tool that address the task of re-engineering classical programs towards an object oriented environment.
In the following, we will describe the general architecture of the tool, the structure of the repository the tool is based upon, the logic of the re-engineering process.

3.1 - Generalities

In our approach, we put much emphasis on the *data*, taking the identification of the relevant data structures as a first step. The motivation is twofold: on one hand, it is easier to perform a reverse engineering on data than on procedures, on the other hand, we may think that if the objects exist, they must be reflected in some data structure. This last point is evident when we are concerned with database applications.
A fundamental aspect is to *capture the semantics* of the existing programs, and this process cannot be accomplished in a completely automatic way, because the access to informal documentation may be necessary, or "tricky" code can make obscure the underlying design issues. As a consequence, a stage of refinement will be necessary in which *human intervention* or heuristically guided search procedures improve the candidate objects.
The general architecture of TROOP is shown in fig. 1, which clearly identify the importance of a central repository where the knowledge deduced from the analysis of the existing software and other information sources can be stored.
The content of the repository will be described in the next paragraph.
The Static Code Analyser is currently under development in the Software Engineering and Applications Laboratory at CNUCE-CNR.
Diagram Server[1] is in charge of displaying graphs described by a formal description, and permits their interactive manipulation by the user. The representations will be

[1] Diagram Server is a tool developed in the context of the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo by Dipartimento di Informatica e Sistemistica - Università di Roma "La Sapienza" ([Di Battista91]).

generated by the ReBuild (Representation Builder) module which makes use of the algorithms reported in the literature, as described in [Signore92b].
The ReComp (Representation Compasrator) module basics will be briefly sketched in the paragraph 3.3.
As far as the Information Retrieval aspect is concerned, we will adopt a document vector space model, where each “document” is identified by a set of weighted keywords, selected from a classification scheme. The classification scheme will be displayed to the user, that will be allowed to navigate through it and choose the right terms, in a way similar to that described in [Signore92c].
The user interface will be developed in a windowed environment and will offer some hypertextual capabilities.

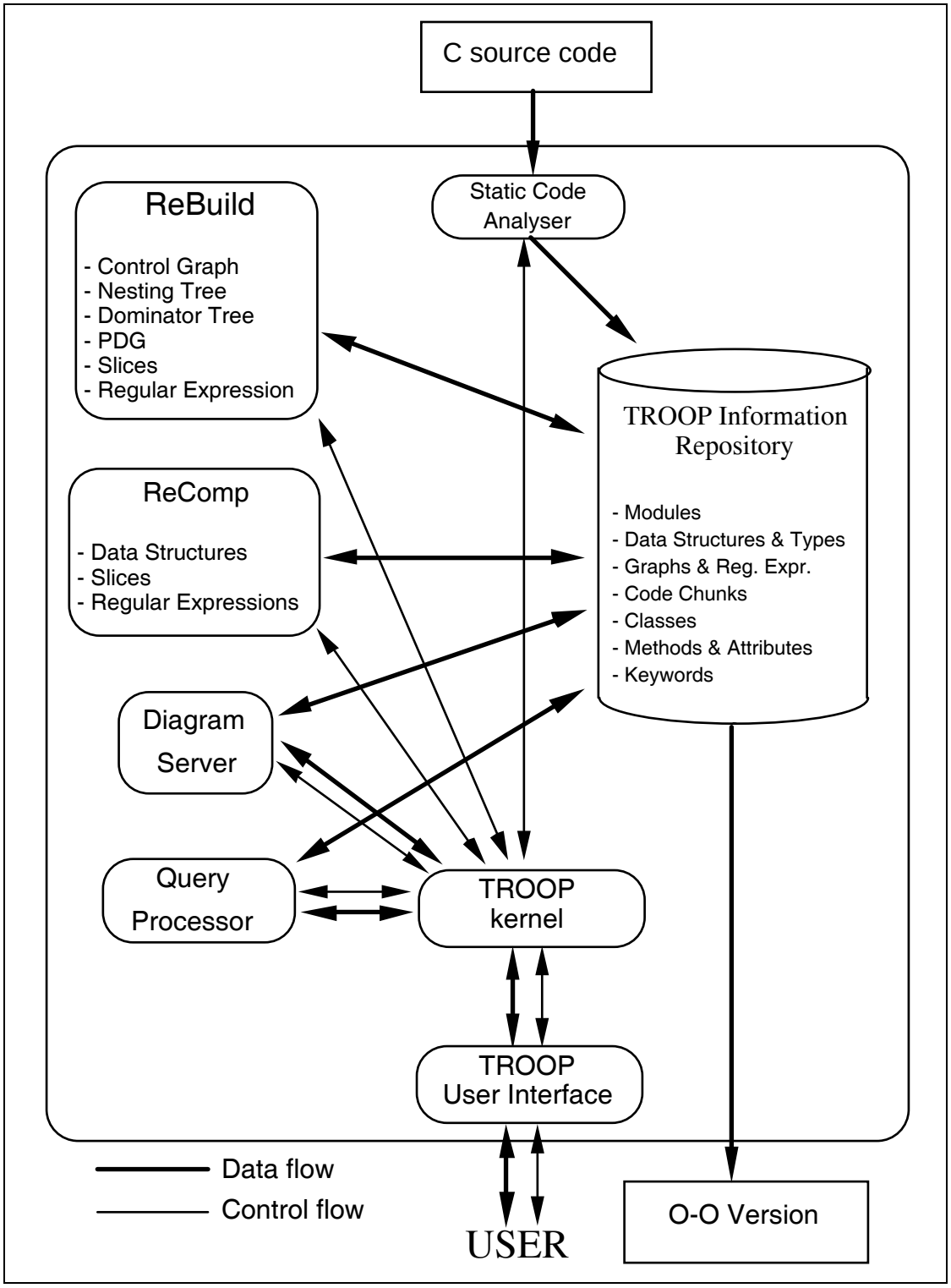

Fig. 1 - The general architecture of TROOP.

3.2 - T.I.R.: the TROOP Information Repository

Even if every software engineering tool possess his own repository, and some large repositories are presently claimed to be available (i.e. IBM Repository or Digital

CDD/Repository), it must be noted that they are either tightly coupled with specific tools, or under constant evolution. Therefore, we preferred to implement by ourselves a repository that could be able to accept and manage all the information we need in the re-enginering process. In this sense, our repository may act as an intermediate level repository.

At present stage, we make use of a standard relational DBMS (Sybase for Sun/OS) even if we are also considering the possibility of migration to an O-O DBMS, that should give a better and more coherent support to the management of the items stored in the repository.

In the following we will conform to the terminology adopted by Eiffel ([Meyer88]).

A rough graphical representation of the conceptual schema of T.I.R., whose structure is currently in an evolutionary stage, is in figure 2. It is evident that we can identify two main sets of entities, those that describe the classical environment (Programs, Modules, Data structures, Code chunks, etc.) and those pertaining to the object oriented perspective (Classes, Attributes, Methods). The Keywords can be used to characterise both the Code chunks and the Methods.

Some of the entities and relationships are obvious, and will not be described in detail. The description of some of the most relevant entities and relationships follows.

- *Code chunks*

  Are pieces of code resulting from the slicing process on the modules.

- *Representations*

  Are the representations of the structure of a Code chunk or of a Module, both as a graph (Program Dependence Graph, Nesting tree, Control graph, Dominators tree) and as regular expression.

- *Classes* and *Attributes*

  Are the O-O classes and attributes that have a representation in terms of Data structures in the conventional programs. The two unary relationships involving Classes model the *inheritance* and *use* relationships in the O-O programming environment.

- *Types*

  Model the conventional types (int, char, struct, array, etc.). The unary relationship involving Types model the subtype relationship.

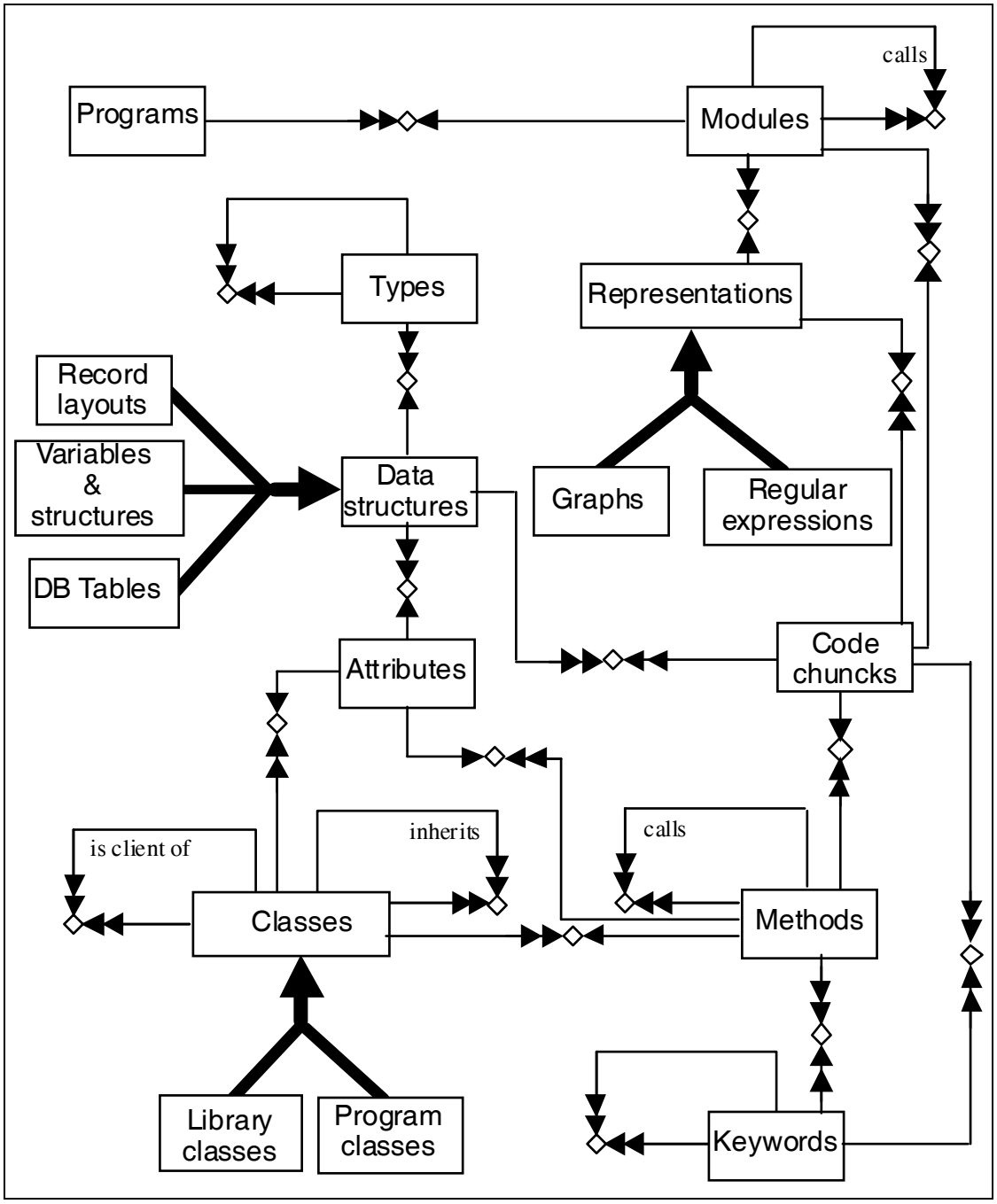

Fig. 2 - The conceptual schema of T.I.R.

3.3 - Description of the re-engineering process

The re-engineering process is concerned both with data and procedures. The whole process starts from the static analysis of the data the programs are managing. Subsequently, the procedures are analysed on the basis of the data they are accessing and manipulating. This constitutes a first criterion for the identification of "tentative" methods. In the following we describe in more detail the various steps.

Objects and fields

1. The very first step consists in the identification of the data structures used by different modules of the existing programs. In this step, we identify the global variables, the record description structures and the data structures that are most used as actual parameters.

When analysing database applications, it is quite easy to identify objects that have been mapped onto database structures. However, the constraints imposed by the relational DBMS force the implementation of repeating attributes as separate tables, and the mapping of complex relationships onto ad hoc link tables. This can lead to the identification of "spurious' objects. The ambiguities must be solved by human intervention.

2. The inheritance hierarchies can be established on the basis of a type classification based on characteristics like access method, scanning, storage ([Meyer90]).

Methods

1. The modules can be arranged taking into account:

- their size, expressed as Lines Of Code (LOC);
- their depth in the calling tree, i.e. their level in the Structure Chart;
- reuse frequency, i.e. the number of times the module is invoked by other modules, in respect to the total number of calls.

2. On the basis of the variables identified in the object identification phase, we proceed to the *slicing* of the modules, starting from the modules of limited size, low level in the Structure Chart, and mostly frequently invoked.

2a. For each connected graph we consider the possibility of identifying the corresponding code as a *method*. Therefore we assign to it a *name*, a set of *keywords*, the name of the *"class"* it is operating on. The keyword are extracted from a faceted classification a browser can graphically display and navigate through. The class is the potential class identified in the object identification phase.

In addition, we identify the *constraints*, representing them as a set of pre and post-conditions.

2b. Subsequently we proceed to the rebuilding of the program, expressing it by means of the identified components. In this step, a restructuring can take place.

2c. When considering modules at higher levels in the Structure Chart, attention is paid to the identification of possible cases of generalisation.

2b. In the last step, we consider the similarities between the potential methods making use of regular expressions where particular substrings are identified by a single label that gives information about its functionalities. Therefore the different methods are compared on the basis of both their syntactic structure and their semantics. In this process, we will make use of the techniques developed in the context of the Information Retrieval area.

4 - Conclusions

In this paper, after exposing the state of art of the re-engineering disciplines, the authors have discussed about the features of the particular repositories which are fundamental in such software engineering activities. Further the reasons that make the re-engineering towards an object-oriented environment an interesting issue have presented.
To accomplish this task, we have sketched the architecture of a re-engineering tool (TROOP) that relies on a central repository, that contains information pertaining to both the traditional like modules, data structures as well as the object oriented target environment like classes, attributes and methods. The main steps and the areas where a human intervention is required have been described.

References

[Alabiso88] Alabiso M.: *Transformation of Data Flow Analysis Models to Object Oriented Design*, Proceedings of OOPSLA' 88, September 25-30, 1988

[Basili90] Basili V.R.: *Viewing Maintenance as Reuse-Oriented Software Development* , IEEE Software (January 1990), pp. 19-26

[Basili92] Basili V.R., Caldiera G., Cantone G.: *A Reference Architecture for the Component Factory*, ACM Transactions on Software Engineering and Methodology, Vol. 1, N. 1 (January 1992), pp. 53-80

[Biggerstaff89] Biggerstaff T.J.: *Design Recovery for maintenance and Reuse* ; IEEE Computer, (July 1989)

[Burton87] Burton B.A., Wienk Aragon R., Bailey S.A., Koehler K.D., Mayes L.A.: *The Reusable Software Library*; IEEE Software, (July 1987), pp. 25-32

[Chikofsky90] Chikofsky E.J., Cross II J.H.: *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software (January 1990)

[Cimitile91] Cimitile A.: *Re-Use Re-Engineering: the RE^2 project*, Proc. of Workshop on Reverse Engineering, Portici, Naples, Italy, December 11 1991

[Di Battista91] Di Battista G.: *A Client-Server Architecture for Constructing Diagrammatic Interfaces,* Proc. of Workshop on Reverse Engineering, Portici, Naples, Italy, December 11 1991

[Edwards90] Edwards J.M., Henderson-Sellers B.: *The Object-Oriented Systems Life Cycle*, ACM Communications, Vol. 33, N. 9 (September 1990)

[Helm91] Helm R., Maarek Y.S.: *Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries* , Proceedings of OOPSLA' 91, pp. 47-61

[Jacobson91] Jacobson I., Lindström F.: *Re-engineering of old systems to an object-oriented architecture*, Proceedings of OOPSLA' 91

[Liu90] Liu S-S., Wilde N.: *Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery*, IEEE Proceedings of IEEE Conference on Software Maintenance, San Diego, November 26-29, 1990

[Meyer88] Meyer B.: *Object Oriented Software Construction*, Prentice Hall International (1988)

[Prieto-Diaz91] Prieto-Diaz R.: *Implementing Faceted Classification for Software Reuse,* Communications of the ACM, Vol 34, N. 5 (May 1991)

[Sedes92] Sedes F.: *A Hypertext Information System for Reusable Software Component Retrieval*, DEXA'92 - Database and Expert Systems Application, Proceedings of the International Conference in Valencia, Spain, 2-4 September 1992, (Tjoa A.M., Ramos I., Eds.) Springer Verlag Wien New York, ISBN 3-211-82400-6, pp. 457-462

[Signore92a] Signore O., Loffredo M.: *Reverse, re-engineering e riuso: tre discipline connesse alla manutenzione*, Technical Report, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, N. 8/34 (October 1992)

[Signore92b] Signore O., Loffredo M.:*Some issues on Object Oriented Re-Engineering*, Proceedings of ERCIM Workshop on Methods and

Tools for Software Reuse, Heraklion, Crete, Greece, October 29-30 1992, pp. 243-265

[Signore92c] Signore O., Garibaldi A.M., Greco M.: *Proteus: a concept browsing interface towards conventional Information Retrieval Systems*, DEXA'92 - Database and Expert Systems Application, Proceedings of the International Conference in Valencia, Spain, 2-4 September 1992, (Tjoa A.M., Ramos I., Eds.) Springer Verlag Wien New York, ISBN 3-211-82400-6, pp. 149-154