EPM/RT-94/25

# Transformations in Reengineering Techniques

Germinal Boloix
Pierre N. Robillard

Département de génie électrique
et génie informatique

gratuit

École Polytechnique de Montréal
Novembre 1994

Dépôt légal, novembre 1993
Bibliothèque nationale du Québec
Bibliothèque nationale du Canada

Pour se procurer une copie de ce document, s'adresser:

Les Éditions de l'École Polytechnique
École Polytechnique de Montréal
Case postale 6079, succ. Centre-ville
Montréal, (Québec) H3C 3A7
Téléphone: (514) 340-4473
Télécopie: (514) 340-3734

Compter 0.10 $ par page et ajouter 3,00 $ pour la couverture, les frais de poste et la manutention. Régler en dollars canadiens par chèque ou mandat-poste au nom de l'École Polytechnique de Montréal.

Nous n'honorerons que les commandes accompagnées d'un paiement, sauf s'il y a eu entente préalable dans le cas d'établissements d'enseignement, de sociétés ou d'organismes canadiens.

# Transformations in Reengineering Techniques

**Germinal Boloix, Pierre N. Robillard**
**3 November 1994**
**Ecole Polytechnique de Montréal**
**Département de Génie Electrique et de Génie Informatique**
**C.P. 6079 succ Centre Ville**
**Montréal, Québec H3C 3A7**
**Tel. (514) 340-4031, 340-4238 - Fax. (514) 340-3240**
**boloix@rgl.polymtl.ca**
**pnr@rgl.polymtl.ca**

## Abstract

Reengineering techniques (REs) seek to understand systems by extracting knowledge from source code. This knowledge is represented by different views and different layers of abstraction. By studying REs, it is possible to determine what knowledge is required and to determine why it was not available in the first place. This understanding of REs should provide important insights into the way new systems should be developed.

In this paper we analyze several REs using a transformational approach for purposes of identifying the characteristics of REs. By formally specifying the characteristics of views (or environments) and the correspondance among these views at different layers of interpretation, it is possible to identify the knowledge manipulated by reengineering techniques. Each environment (e.g., code, dataflow graph, structure chart) is formally represented using an object-oriented model to define the objects, its associations and the operations upon them. Correspondences among environments are defined using a transformational approach.

## 1.0 Introduction

Reengineering (RE) techniques (e.g., reengineering, reverse engineering, restructuring, remodularization) manipulate different types of knowledge, abstracted from the implementation, to generate higher-level views understandable to humans. From these views, a forward-engineering process may take place. Some reengineering approaches claim the system's functionality remains unchanged. Other approaches, like business-process reengineering, may require a complete overhaul of the systems to conform to new models of the business.

The user's view of systems is functional, and related to the application domain. The developer's view of systems requires knowledge of both the application domain and software techniques. Matching functionality to programming constructs aids during software evolution activities. Corrections or enhancements are performed at the programming level, but may impact the application domain. The goal is to define a formal mapping from programming constructs to application domain functionality.

REs utilize different approaches to generate software representations. Composition, decomposition and transformation are some mechanisms for managing software representations. Composition is the process of assembling information by finding relevant components and establishing their relationships. Decomposition creates objects and relationships derived from existing components. Transformation maps components among different representations. Formalizing the reengineering process is a step towards understanding precisely what type of knowledge is required.

Our approach is to analyze several reengineering techniques to identify this knowledge, from implementation to high-level representations of systems. Implementation concepts are matched to layers concepts to establish the knowledge manipulated by each reengineering technique. Each environment (e.g., code, abstract syntax trees, dataflow graphs, structure charts) is formally represented using an object-oriented model to define the elements, its relationships and the available operations. Correspondences among environments are defined to establish the transformations steps. This approach has already been analyzed for forward-engineering activities [BST92], and integrates common concepts from reengineering and forward-engineering approaches.

We would like to identify what knowledge is required to understand a system or a piece of code. Some knowledge is present in the source code, but hidden from the point of view of understandability. Knowledge evolving during development or maintenance which was not captured has to be captured with external assistance. Knowledge about the programming language, the operating system, the problem domain, the solution space, the evolution of the system, as well as an understanding of the different types of software representations at high layers of abstraction, is required. The knowledge abstraction problem [A87] consists in differentiating domain knowledge from implementation knowledge through layers of abstraction.

The paper is organized as follows. Section 2 presents a summary of reengineering techniques. Section 3 introduces formal representations of environments and their transformations. Section 4 summarizes the knowledge captured from the implementation and the knowledge produced at different abstraction layers. Section 5 presents the knowledge required and produced by each reengineering technique. Finally, section 6 gives some conclusions and suggest future areas of research.

## 2.0 Reengineering Techniques

Reengineering techniques (REs) are oriented towards improving maintenance through an understanding of existing source code and the generation of a better software. The object of these efforts is to reduce maintenance costs and increase maintenance productivity by improving software quality and reducing software complexity. The main objectives of REs are to understand and improve software (i.e., make it more adaptable to change), and to capture, preserve, and extend knowledge about software. The process of creating or reconstructing software has to be considered together with the characteristics of the software itself, if improvements of orders of magnitude are to be expected. Design decisions and their rationale are documented to help understand the reasons behind the alternatives. REs seek to express knowledge of programs and data structures in terms closer to the problem domain, understandable by developers and maintainers, in order to optimally manage system evolution.

### 2.1 Summary of reengineering techniques

A summary of reenginering technique definitions is presented which identifies its main characteristics (more detail can be found in Arnold [A93]). In most cases REs create new systems without changing their functionality; changes in functionality should be made after the system has been reengineered. Other RE techniques only create high-level representations of the system for purposes of understandability. Finally, business process reengineering may have a major impact on software because the business itself is affected.

Reengineering comprises reconstruction of software to adapt it to new environmental conditions. It changes the underlying technology of a system without affecting its overall function, basically updating the technology rather than improving functionality. It requires uncovering the essential software requirements and the user's underlying model. It examines and alters the system to reconstitute it in a new form, and to finally implement the new form.

Examples of conversion approaches in reengineering are:

- Assembler to third-generation language
- Structured Cobol to open-system environment
- Cobol-74 to Cobol-85
- Fortran to Ada
- Ultra to Pascal
- Convert from old DBMS and file structures to new relational DBMS

- Conversion from Univac to IBM 370

Reverse engineering generates software representations that help in understanding it or facilitate its processing. It is a backward process from code to a high-level representation similar to that of a design document. The process of reverse engineering analyzes a system to identify the system's components and their interrelationships, and to create representations of the system in another form at a higher layer of abstraction. This transformation goes from low-level technology considerations (i.e., code) to high-level enterprise model representations (i.e., business rules).

Restructuring is the process of changing the source code control structure according to the rules of structured programming without changing external functionality. This transformation is done at the same abstraction layer, preserving the system's external behavior. It implies modifying the software to make it easier to understand and to change, or make it less susceptible to error when future changes are made. A major goal of software restructuring is to preserve or increase software value (e.g., cost savings to users and more productive maintenance activities). For large systems, remodularization is necessary, in addition to restructuring.

Reuse of software involves selecting existing components and adapting them to perform for different environments. Besides code, there are several types of software assets that can be reused: specifications, designs, test data, software documentation and code templates. Reusing a software component may imply concurrent reuse of other components associated with it. Criteria for reusability are the functional usefulness of components, the cost to adapt components in a new system, their environmental independence, and the degree of dependency among components. Experience from past development and maintenance activities can also be reused along with the components, provided that the experience is recorded.

Remodularization is the reorganization of the system's architecture. Similarity among program or system parts is analyzed and a new structure is proposed. As in other REs, the functionality of the system should not be altered. Modules should be logically meaningful to facilitate their association with the application domain.

Design recovery is the process of identifying the high-level design abstractions comprising the system, the domain knowledge and the reasoning behind decisions. The design recovery process looks for large-scale organizational structures (i.e., subsystems, modules, data structures). A domain model of the system is built that captures the abstract design components. Informal information from experts is also captured to document the design. Recovered design abstractions include formal specifications, module breakdown, data abstractions, data flows, program designs (PDL) and informal knowledge about the application and problem domains.

Data reengineering is a system-level process that normalizes data structures and purifies data definitions and values. It provides meaningful, non-redundant data definitions, and valid, consistent data values. It recognizes and eliminates dataflow anomalies. The process can be partly automated. Data restructuring changes the underlying data model. Users want to access data transparently and move applications across hardware platforms.

Business process reengineering requires looking at the fundamental processes of the business from a cross-functional perspective. The objective is to use information technologies to enable a new, more efficient business process, instead of maintaining existing software. Quality, innovation and service may be more important than cost, growth and control. Business process scope is interorganizational, interfunctional, and interpersonal instead of localized. The business vision has to be defined along with the objectives of each process. Some guidelines for improving business processes include capturing information once, and, at the source, organizing around outcomes, not tasks, having those that use the output of the process perform the process, putting decision points where the work is performed, and treating geographically dispersed resources as though they were centralized.

Redocumenting is the process of creating a semantically equivalent representation within the same relative abstraction layer. Alternate views are intended for a human audience. Documentation helps in understanding code, in planning and performing modifications, and in performing testing. Source code documentation is upgraded with in-line, expanded and accurate comments.

The traditional process of forward engineering develops high-level abstractions and logical, implementation-independent designs, up to the physical implementation of a system. It follows a sequence from requirements through design to implementation. Some REs apply techniques of forward engineering once the architecture of the systems has been recovered.

Candidate systems for reengineering are those with code over seven years old, systems with an overly complex structure, code written for a previous generation of hardware and systems implemented with very large modules and systems using inflexible structures (e.g., hard-coded parameters).

## 2.2 Knowledge in reengineering techniques

RE techniques require different types of knowledge to understand the code and its high-level representations. Each representation of software portrays some degree of knowledge. This knowledge is present in the form of elements and their relationships. Different layers of abstraction are required for understanding at any representation layerl. The process being followed to reengineer is also important for evolution activities, as well as knowledge about the technology implementing the system. There is also the knowledge about the principles governing the optimal assembly of elements and their relationships. Finally, there exists the knowledge about the application domain, which may be embedded in the software.

### Global knowledge

Documentation (Understanding)

Legacy systems normally lack documentation because of their evolving history. These systems require external and internal documentation to be understood. The process of understanding a system requires support documentation in the form of manuals (e.g., user, operation, system) or mixed with the source code (e.g., comments). Diagrams showing different representations of the system are additional sources of documentation. Documentation helps to understand code, plan and perform modifications, and perform testing.

Principles (Standards)

Principles governing the optimum arrangement of elements is required to improve software engineering practices. Enforcing standards for systems is also a way of achieving consistency across applications. Maintainers benefit from consistency when working on several systems. Programming standards and style guidelines are used to improve software quality.

## Abstraction

Layers (Levels)

A conceptual model is a logical representation of a system. This model is organized by layers of abstraction. These layers may not coincide with the existing system structure. High-level representations of systems facilitate understanding of the underlying implementation model. From the information available at the implementation layer, higher-level abstractions of the system are built using simplifying transformations. Ideally, the abstraction process tries to recover design and specification documentation to help maintainers during software evolution.

Architecture (Structure)

Portraying the architecture of a system helps to understand its functionality. Systems are partitioned into subsystems, modules, and routines. This partitioning facilitates identification of pieces of code that perform specific functions. Relating system structure to user's functionality facilitates maintenance. The physical architecture of a system is different from its logical architecture, representing the conceptual model of the system.

Views (Visualization)

Different views of software (e.g., structure, data, control, behavior) are required to gain a complete understanding of the system. Source code only provides a textual representation of software, whereas call graphs, control graphs and dataflow graphs are examples of graphical aids. These additional views provide maintainers with understandability.

## Process

Design Rationale (Reasoning)

The evolution of a system involves several design decisions during development and maintenance. These decisions indicate the reason for a particular structure being chosen. Having the rationale behind design decisions helps in understanding the system.

Measurement (Analysis)

Metrics to measure software properties are required to reengineer systems. Through analysis, it is possible to identify hard-to-maintain code that requires reengineering, and choices are made to improve software quality.

Methods (Techniques)

Techniques to represent software at high layers of abstraction are fundamental during reengineering. These methods or techniques are similar to forward-engineering methods.

**Domain**

Function (Application)

Knowledge about the application domain is a key issue in building better systems, supporting the rationale behind decisions and facilitating an understanding of the system. Management and clerical rules put into systems are a source of knowledge about the application domain that may be retrieved from systems for documentation purposes. Reengineering business processes is an attempt to consider larger issues that impact the effectiveness of the business rather than system performance only. Domain experts are a valuable source in documenting existing systems.

Data (Information)

Data is a more stable resource than procedures. Reengineering the data is an important objective in RE techniques. A good choice of data structure facilitates the evolution of systems. Data definition and data value problems abound within existing systems. Dataflow anomalies jeopardize system correctness.

Some data definition problems are cryptic, inconsistently named, use names that are not descriptive, and have inconsistent field names. Some data value problems contain inconsistent default values, lack distinction between valid and missing values, and contain most-significant digits that are truncated.

**Technology**

Tools (Facilities)

Knowledge about software technology is fundamental. Programming languages, virtual machine environments and software tools are some examples of the knowledge required to build, reengineer and maintain software.

Repository (Data Dictionary)

Development of systems around a repository has been suggested as a way to keep documentation updated. A data dictionary keeps track of software definitions, and can be used to document different representations of the system, including application-domain information.

Figure 1 shows a summary of important reengineering knowledge, organized in four dimensions. There is an overall type of knowledge which involves documentation requirements in each dimension and the principles to usefully produce software. One dimension identifies the functionality and application domain of the software. Another dimension identifies issues regarding the software reengineering process, the reengineering rationale and the methods being utilized. The abstraction layers dimension identifies the importance of various layers, different views of the software and its architecture. Finally, the last dimension identifies issues regarding the technology and tools used during reengineering activities and to implement the system.

## 3.0 Formalizing Reengineering Techniques

Our objective is to identify what system knowledge is recovered by REs. Some of this knowledge is present in the source code, but hidden from the point of view of understanding. Other types of knowledge which evolve during development or maintenance but are not captured formally, have to be gathered with external assistance. We want to identify what knowledge is required to understand a system or a piece of code. Knowledge about the programming language, the operating system, the problem domain, the solution space and the evolution of the system, as well as an understanding of the different types of software representations at high layers of abstraction, is required.

Figure 2 presents the transformation approach from code to intermediate representation, and eventually back to code. The transformation process involves defining the correspondences among different environments. The original code is represented by the source environment, at the code or implementation layer. Intermediate representations are high-level views, possibly derived from code, for purposes of understanding. Target environments are represented as code, a new language implementing the system. A repository with all the knowledge of the system as well as the automatic support required to build each environment consistently is required if automatic tools are going to be developed. Metric support contributes to improved software development by identifying complex areas that should be reengineered first.

It is important to mention that in all reengineering techniques human participation is required. We represent this in the form of an oracle that provides the information required to perform the correspondence from one environment to another. The figure of the oracle can also be identified with the business process model that impacts the characteristics of the system.

## 3.1 Object-oriented approach

REs are closely related to forward-engineering approaches. Both are integrated for purposes of the development and maintenance of software. To identify the knowledge manipulated by REs, an approach based on the specification of REs is proposed. The approach defines an environment as a representation of software using the same vocabulary. Defining transformations among different environments [BST92] helps to identify the correspondences among environments using the same or different vocabulary.

An object-oriented approach defining environments (i.e., representations of software) is presented [KR94]. The objects to be manipulated, their associations and the operations upon them are the elements needed to establish correspondences among environments. Other powerful characteristics of the object-oriented approach, such as inheritance, are useful once common patterns of transformations are identified, for reuse purposes.

### Environment

An environment is the representation of software using a specific vocabulary. Code would be an environment at the implementation layer, structure charts would be an environment at the architecture layer.

### Objects

Objects are the basic building blocks for defining the environment vocabulary. Code objects include operators and operands. Structure chart objects include modules, user interfaces, parameters and files.

### Associations

Objects in the environments are not isolated, they maintain relationships with other objects. These relationahips are prespecified to define the interactions among objects. In code, the locations of operators and operands are fixed by grammar rules, and some statements have to be placed according to preestablished patterns. In structure charts, modules are allowed to call other modules, but user interfaces are not associated with similar user interface objects, for example. By establishing associations during the definition of an environment, constraints on permitted relationships are automatically enforced.

### Operations

When creating environments, additional constraints may be imposed on the characteristics of objects and their associations (e.g., graphical location constraints). When transforming environments from one abstraction layer to another, mapping operations are defined to relate objects from different environments.

A contract-based approach is being proposed to specify these transformations. Specifying mappings among environments requires defining what must be true before performing the mapping (precondition), and what must be true after performing the mapping (postcondition). These conditions must be true relative to what must be true for all operations (invariants).

**Aggregation**

Aggregation is an important aspect of environment definitions as aggregation relates to abstraction layer concepts. Aggregates may be composed of several objects and associations, by themselves representing new objects.

## 3.2 Example of environments and transformations

An example of the steps required to visualize the correspondence between a code environment, at the implementation layer, and a structure chart environment, at the architectural layer, is described. Each environment is defined using the object-oriented approach and the correspondence between one environment to the other is specified using the same formalism.

### 3.2.1 Code-layer environment

At the code layer, language characteristics are a major concern. The Abstract Syntax Tree (AST) is another type of environment very close to code. Let us give a summary of the objects and their associations required at the code layer.

Objects:

- Operators (e.g., mathematic operators, boolean operators, etc.)
- Operands (e.g., variables, labels, etc.)
- Statements (aggregates of operators and operands)
- Blocks (aggregates of statements)
- Procedures (aggregates of statements and/or blocks)
- Programs (aggregates of procedures and/or blocks and/or statements)

Associations:

- language grammar orderings
- invoking procedures

Operations:

- constraints on the language syntax (location of elements)

### 3.2.2 Architectural-layer environment

At the architectural layer, structure charts define the structure of the system in terms of modules and their interaction. Let us present an example of an environment definition for structure charts.

Objects:

- Modules
- Parameters
- Interfaces
- Files

Associations:

- Communication with module (Interface, Module) /*relationship with user*/
- Call Module (Module, Module) /*relationship between two modules*/
- Access (Module, File) /*relationship with database*/

Operations:

- Constraints on allowable connections among objects

### 3.2.3 Correspondance from code to architecture

Mapping rules from the source code environment to the architecture environment are specified using contract-based operations. Examples of correspondences or transformations are:

Description {Define Module}

Parameters {Procedure (Code environment); Module (Structure Chart environment)}

Invariant {Procedure exists}

Precondition {Module (corresponding to Procedure) does not exist}

Postcondition {Module (corresponding to Procedure) exists}

------------------------------------------------------------------------

Description {Define Call}

Parameters {Procedure, Invoke (Code environment); Module (Structure Chart)]

Invariant {Modules exist; Procedures exist; Procedure is invoked}

Precondition {Module does not Call same Module}

Postcondition {Module Calls Module}

# 4.0 Implementation and Abstraction Layer Concepts

Identifying the main objects to be manipulated in different environments, from the implementation environment up to the application domain, would help in identifying the different types of knowledge required for each RE.

## 4.1 Implementation Concepts

Programs are textual representations of data and logic expressed in a particular language. Syntactically, a program is a sequence of text strings, but semantically they contain many types of concepts, including language and abstract concepts. Language concepts are variables, declarations, modules, statements, and so on, that are defined by the coding language. Statements are formed by operators and operands. Constants, variables, labels, and arithmetic and logical operators are some examples of operators and operands. Assignments, calls, declarations, comments and control statements are some examples of statements.

A program can be understood at different layers of abstraction. The lexical level presents the operators and operands. The syntactical level presents the statements and the structures (e.g., data structures and control structures). The style level characterizes a program according to similar patterns of coding such as structured programming. Finally, the semantic level identifies patterns of computation such as counters, enumerators and algorithms.

### 4.1.1 Lexical level

Language concepts are textual in nature and tend to be specific to a particular language syntax.

**operators**

Different types of operations are permitted in a language. Through symbols or key words, operators are defined in the language.

**operands**

The data being transformed in a program are represented by variables which identify their current value. Some symbols represent constants that keep the same value over the computations.

### 4.1.2 Syntactical level

Operators and aperands are arranged together in statements. A series of statements is associated in a structure or block.

**statement**

Several statements are available to build the logic of programs. Assignments, declarations, comments, control statements and calls to subroutines are some examples.

**structure**

A structure is a sequence of statements. A structure is not a compilable unit in itself. A structure has one entry point and one exit point.

**procedure**

A procedure is an element that can be compiled, but its execution only makes sense within the context of a program. Procedures are aggregates of software elements (i.e., statements, variables, structures). There are procedures that are developed anew for a system or reused from another system or library. There are procedures that are called from system libraries (e.g., sort, search).

**program**

A program is an element that can also be compiled. Programs are also aggregates of software elements, as in the case of a procedure.

### 4.1.3 Style level

Programs can be written using different patterns of arrangements. A program written using go-to statements would be different in style from one using only structured constructs like 'if-then' and 'do-until'.

### 4.1.4 Semantic level

The interpretation of the programming logic is defined by the algorithms and their computation objectives.

### 4.1.5 General Implementation Concepts

Analyzing several programming languages, it is possible to generalize a series of program concepts at the implementation layer.

**Documentation and Data (lexical level)**

There are different mechanisms to document at the implementation layer. Comments in the code are used for several purposes (e.g., domain knowledge or reasoning about the logic in the program). Names of variables also offer a mechanism for documentation of programs. Finally, external documentation (e.g., specifications and system manuals) represents a useful source of documentation.

**Control and Procedures (syntactical level)**

Control considerations in programs are primarily associated with conditional statements and their corresponding control structures. Other types of statements like a group of sequential statements or statements invoking procedures, are associated with control as well.

A procedure represents a conceptual cohesive unit in a programming language. It may be associated with a specific function. A module is usually associated with a procedure or routine, but it may also be associated with an object in an object-oriented programming language.

**Program (semantical level)**

A program represents a set of statements that implements a computation. It may be associated with either an algorithm or a function.

## 4.2 Abstraction Layer Concepts

Program concepts are constructs of symbol sequences in a particular programming language with a preestablished syntax. To understand a program, syntax and semantic knowledge of the language is required. The process of understanding involves an abstraction mechanism from low-level representations to high-level representations up to the application domain.

Abstraction layer concepts represent language-independent ideas of computation and problem solving methods. Abstraction layer concepts can be classified into programming concepts, architectural concepts, behavioral concepts, functional concepts and domain concepts. Abstraction layer concepts may not be localized, and they do not necessarily occupy consecutive sections of code.

**Programming: Data structures, algorithms, strategy**

Knowledge of programming language syntax is the first aspect to consider at this layer. Programming concepts included at this layer are coding strategies, data structures and algorithms [HN90]. Stereotyped code patterns of common programming strategies can be recognized from the code (e.g., accumulators, enumerators, data movement).

**data structures**

Declarations of constants, variables, records and their operations are some examples of data structures that can be abstracted away from the language. There are also dynamic data structures, like stacks, queues and graphs, that can be recognized from the code.

**strategy**

Knowledge of stereotyped code patterns of common programming strategies is identified. Strategies include accumulators, enumerators and data movement.

### algorithms

Standard algorithms to solve common problems like mathematical computation, searching and sorting, can be abstracted from the source code.

Examples of programming-level representations, besides the code itself, are abstract syntax trees and symbol tables of program tokens, dataflow graphs, control-flow graphs and dependency graphs.

## Architecture: Components, interrelations

The structural-level view abstracts a program's language-dependent details to reveal its structure from different perspectives. The result is an explicit representation of the dependencies among program components. Examples of architecture layer views are structure charts and data architectures.

### system

A system is the highest representation level of its components. The components combine in a variety of ways to transform information. Presmann [P92] defines a system as 'A set or arrangement of elements that are organized to accomplish some method, procedure or control by processing information'. It is the application software and its interfaces that we are describing, together with its databases. The application software is composed of procedures that perform the functions of the system.

### subsystem

Subsystems are the next hierarchical level representation of the system. Functionality is divided among subsystems. Each subsystem implements functionality by organizing behavior into sets of programs.

### programs

Programs are sets of procedures that accomplish a purpose. Programs are composed of statements, variables, control structures, blocks, calls to routines and the logic of the routines themselves. A program is a component that can be compiled and executed independently.

Example of representations at the architectural layer are structure charts and entity-relationship diagrams.

## Behavior: Events and transactions

Dynamic considerations during the execution of the system are important at this layer. The triggering of events in many software applications involves data or control items that affect the functions performed by the software. Transaction flow is characterized by data moving along the incoming paths of the system and converted into a transaction. The transaction is evaluated and flow along many action paths is initiated.

Examples of such representations are state transition diagrams and petri nets.

### Function: Logic

The functional-level view relates pieces of programs to their functions to reveal the logical (as opposed to the syntactical or structural) relations among them. Each component of the function-level view is an abstract layer representation of a class of functionally equivalent, but structurally different, implementations.

Examples of such representations are data flow diagrams (DFDs).

### Domain: Business rules

The domain-level view further abstracts the function-level view by replacing its algorithmic nature with concepts specific to the application domain. Business rules represent the view closest to the problem domain, reflecting the user's view of the system.

An example of such a representation is the BIRTS model (Business, Information, Rules, Transactions and Scenarios) [G94].

## 5.0 Knowledge manipulated by REs

The purpose of this section is to identify the type of knowledge manipulated by reengineering techniques. These techniques analyze the code, in a reverse-oriented approach, using part of or all the programming concepts available, subsequently producing a new system using forward-engineering techniques.

We have analyzed some restructuring techniques and identified those concepts being manipulated. Other reengineering techniques are analyzed in the same way and a summary of findings is produced.

## 5.1 Restructuring

'Software restructuring is the modification of software to make it easier to understand and to change, or less susceptible to error when future changes are made.' [A93] There are different techniques associated with the term restructuring. We have analyzed code-oriented techniques for purposes of identifying their knowledge manipulation.

### Writing-style techniques

Writing-style techniques involve esthetic changes to the code for purposes of human understanding. The format of the source program is improved without changing the functionality of the program. Changes to variable names, the introduction of more comments in the code are some additional concerns in this type of restructuring.

- Pretty printing
- Code formatting

- Code standardization

### Coding-style techniques

Control flow restructuring changes the structure of the program without affecting its external functionality. New statements are introduced, new variables are defined, and additional comments may be neccessary. Several restructuring techniques have been proposed [Y75].

- Goto-less approach (Bohm & Jacopini)
- State-variable approach (Ashcroft & Manna)
- Duplication-of-code approach (Yourdon)
- Boolean-flag approach (Yourdon)
- Case-statement approach (Linger & Mills)
- Graph-theory approach (Baker)

### Packages/Reusable code

This technique replaces whole sections of code with new code, normally at a high-level of program concepts such as new procedures or programs.

### Data restructuring

Data restructuring affects basically the data aspect of programs, but may also involve changes to current statements. Documentation has to be updated to reflect those changes in data structure.

Figure 3 presents the association of abstract layer/ implementation concepts to identify each restructuring technique in this context.

## 5.2 Reengineering Techniques

Each reengineering technique manipulates different types of knowledge. Figure 4 presents a classifying scheme according to the abstract layer/ implementation concepts. Some reengineering techniques cover several abstraction layers (e.g., reverse engineering, design recovery, redocumentation) whereas others apply at the same abstraction layer (e.g., restructuring, remodularization).

As described in the last section, coding style techniques require knowledge related to documentation, data and control. These techniques analyze knowledge at the intramodule level. It is possible to assert that these techniques manipulate knowledge only at the programming layer, as external functionality is not affected.

Data restructuring affects data definition aspects, including documentation. Depending on the impact of data restructuring, some statements may need to be rewritten, possibly affecting the control aspect of the program. The layers involved are the programming and the architecture layers, the latter concerning data architecture.

Reverse engineering manipulates knowledge of data, control, modules and program to produce additional documentation for the system in the form of diagrams showing the program structure. This technique primarily impacts the architectural layer.

Remodularization generates new system architecture, normally changing statements at the programming layer. The external function of the system should stay basically untouched. The layer of knowledge manipulation corresponds to the programming and architectural layers.

Reengineering is a technique that involves technology changes for the system implemented. It may affect several implementation concepts: documentation, data, control,procedure and program. It involves the programming, architecture and behavior layers.

Design recovery is a technique that manipulates concepts at several layers: architecture, behavior and function. This technique is similar to reverse engineering, but its scope is broader.

Process reengineering extracts knowledge for purposes of identifying business rules at the domain layer. It requires knowledge from the documentation, data, control, procedure and program concepts. Business Process Reengineering may impact the software internal characteristics once decisions to modify the business have been made.

Redocumentation affects all layers of views, as documentation can be produced at any layer for any vocabulary. Documentation, data, control, procedures and program concepts are affected.

## 6.0 Conclusions and further research

We have analyzed several reengineering techniques to determine the type of knowledge manipulated by each of them. During this process, we discovered the magnitude of the problem, which involves knowledge serving different purposes: software knowledge and domain knowledge. It also involves differences among environments with different types of vocabulary or views (i.e., abstraction layers) and differences among representations within the same environment.

We have proposed an approach to identify the knowledge manipulated by different reengineering techniques. The approach utilizes formal specifications of the environments, starting at the implementation layer with the code, following by some intermediate view using a different vocabulary, and eventually implementing again the software using another technology. The formal specification model is based on the object-oriented approach. This approach has the same advantages of current specifications techniques, which permit formal analysis and understanding.

Our objective has been to determine what type of knowledge is required for each of the views, from code to application domain. Reengineering techniques, together with forward-engineering techniques offer a medium in which to analyze the type of concepts manipulated at each layer of abstraction. We are in the process of studying several reengineering techniques for purposes of classifying them according to our approach. This study should indicate possible improvements in our model. We expect to contribute to a better understanding of reengineering techniques in particular, and to software engineering practice in general.

By studying REs, it is possible to determine what knowledge is required and to determine why it was not available in the first place. This understanding of REs should provide important insights into the way new systems should be developed.

## References

[A87] Abbott, R.J. 'Knowledge Abstraction', Communications of the ACM, Vol. 30, No. 8, August 1987, pp. 664-671.

[A93] Arnold, S.R. 'Software Reengineering', IEEE Computer Society Press, 1993.

[BST92] Boloix, G.; Sorenson, P.G.; Tremblay, J-P. 'Transformations using a metasystem approach to software development', Software Engineering Journal, November 1992.

[G94] Goti, J.C. 'Business Rules: Their use for Business Operational Analysis', Technical report TR 03.550, IBM, Software Solutions Division, Santa Teresa Laboratory, San Jose, California, April 1994.

[HN90] Harandi, M.T.; Ning, J.Q. 'Knowledge-Based Program Analysis', IEEE Software, January 1990, pp. 74-81.

[KR94] Kilov, H.; Ross, J. 'Information Modeling, an object-oriented approach', Prentice-Hall, Inc., 1994.

[P92] Pressman, R.S. 'Software Engineering: A Practitioners Approach', McGraw-Hill, third edition,1992.

[Y75] Yourdon, E. 'Techniques of Program Structure and Design', Prentice-Hall, Inc., 1975.

## Software Knowledge

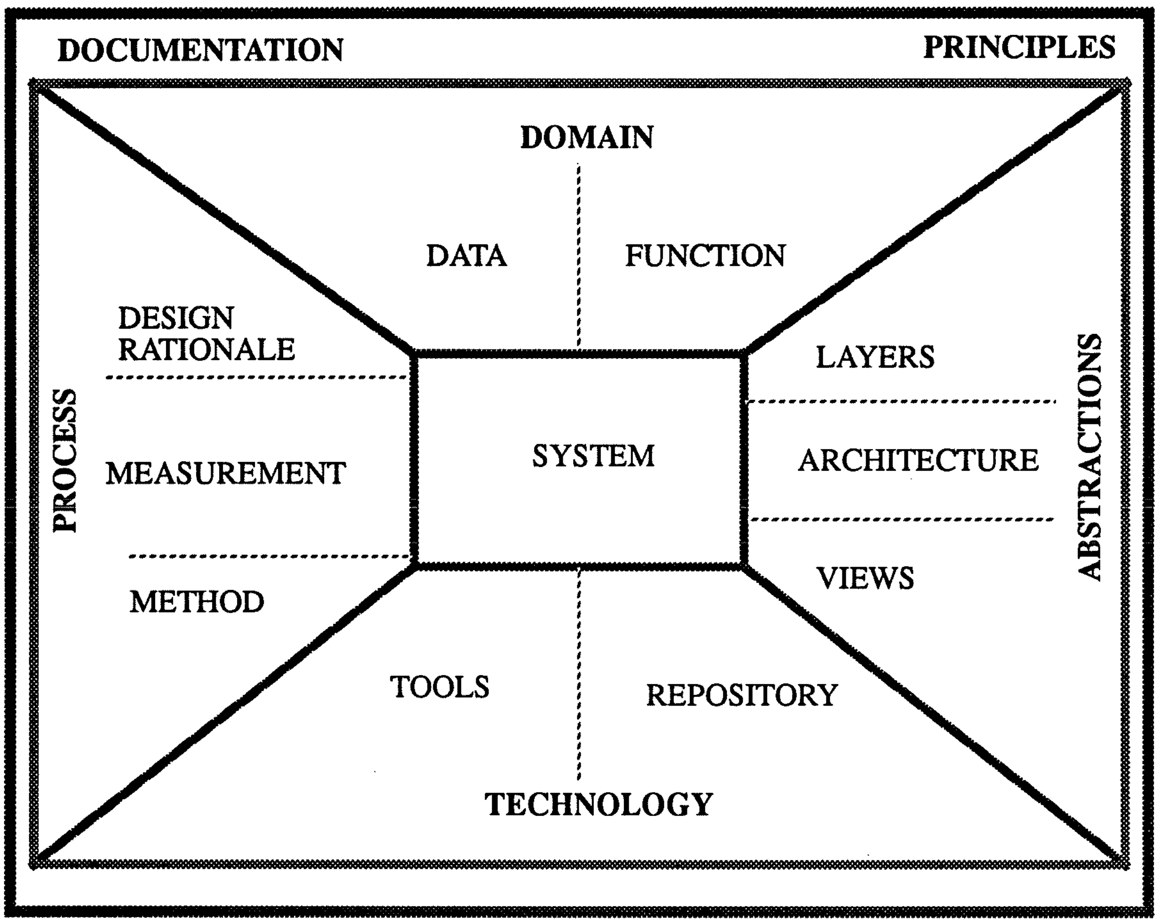
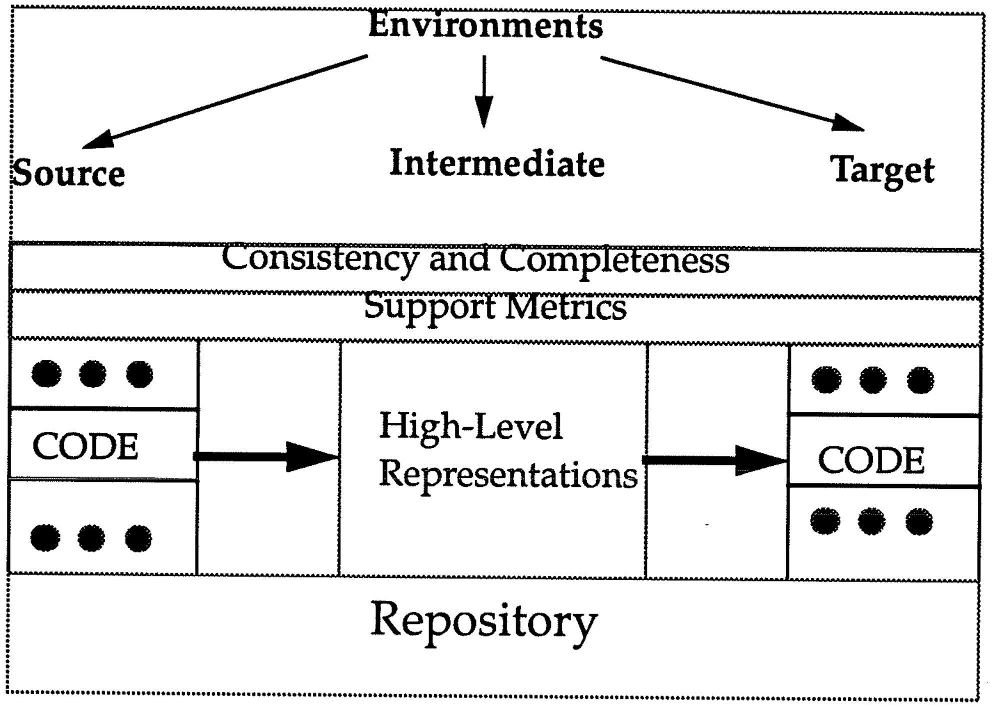


**Figure 1**

## Reengineering Transformations



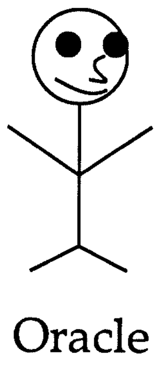


Figure 2

## Restructuring Techniques

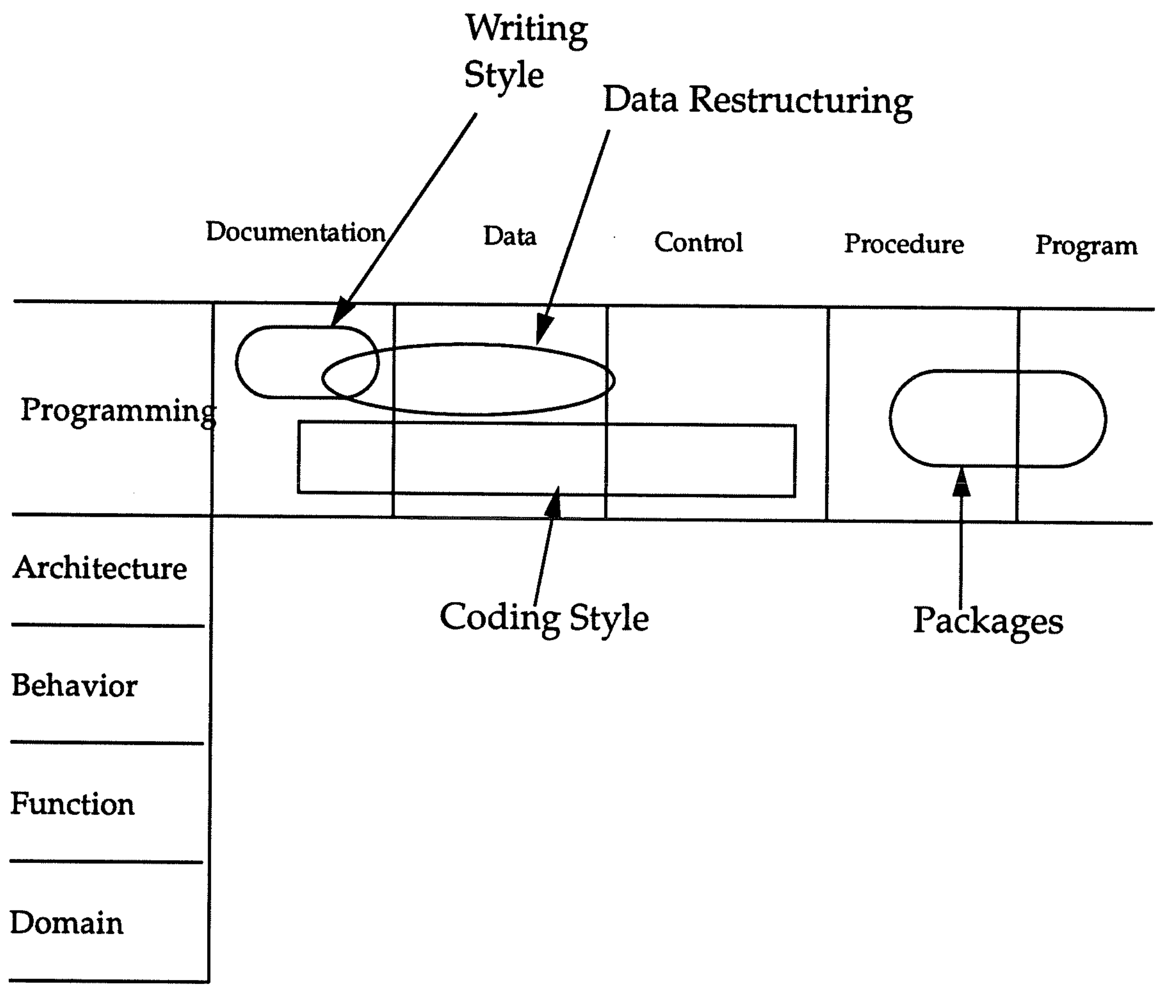


**Figure 3**

## Reengineering Techniques

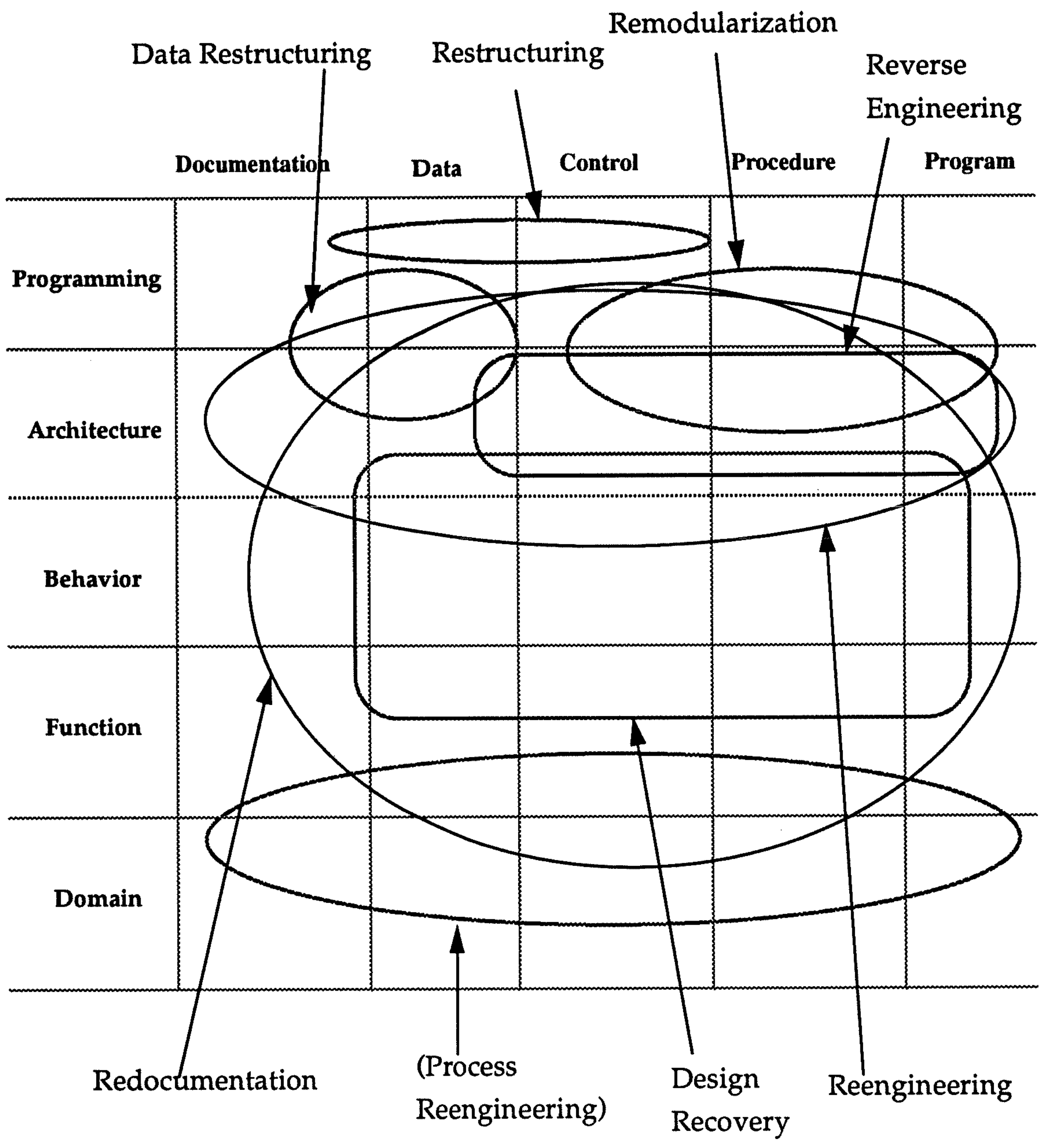


**Figure 4**