



RENAISSANCE

*Method and tool support for the
evolution and re-engineering of legacy systems*

RENAISSANCE FRAMEWORK

©RENAISSANCE Consortium 1997

Version 1.0, First published April 1997

The RENAISSANCE project is partially funded by the European Commission under the Framework Initiative (ESPRIT 22010). The objective of the project is to develop a systematic method to support the re-engineering of legacy systems. Further information about the project is available on the World-Wide-Web at URL:

<http://www.comp.lancs.ac.uk/projects/renaissance/>

The members of the RENAISSANCE Consortium are:

CAP Gemini Innovation (Mr Alain Dineur)

Bâtiment Karélian
7, chemin de la Dhuy
38340 Meylan, FRANCE
Tel: +33 476 76 47 47; Fax: +33 476 76 47 48

CAP Gemini IST (Mr Alain Paoli)

Tour Anjou
33 Quai de Dion Bouton
92814 PUTEAUX Cedex, FRANCE
Tel: +33 1 41 26 63 36; Fax: +33 1 41 26 52 17

debis Systemhaus GEI GmbH (Mr Markus Breuer)

Pascalstraße 14
D-52076 AACHEN, Germany
Phone: +49 2408 943 0; Fax: +49 2408 943 119

INTECS Sistemi S.p. A. (Mr Giancarlo Savoia)

Via Livia Gereschi, 32
56127 PISA, Italy
Tel: +39 50 545 111; Fax: +39 50 545 200

Telesoft S.p. A. (Mr Fabio Mungo)

Via Degli Agrostemi, 30
00040 SANTA PALOMBA (Roma), Italy
Tel: +39 6 710 551; Fax: +39 6 710 553 50

Engineering - Ingegneria Informatica S.p. A. (Mr Dario Avallone)

Via dei Mille, 56
I-00185 ROMA, Italy
Tel: +39 6 522 431; Fax: +39 6 522 432 48

Lancaster University (Prof. Ian Sommerville)

Computing Dept,
Bailrigg, LANCASTER LA1 4YR, UK
Tel: +44 1524 593795; Fax: +44 1524 593608

SINTEF (Prof. Reidar Conradi)

O. S. Bragstads plass 2 F
N-7034 TRONDHEM, Norway
Tel: +47 73 593 444; Fax: +47 73 594 466

Executive Summary

The RENAISSANCE method aims at supporting the reengineering of legacy software systems, that is the transformation of valuable software assets which are difficult to maintain toward new systems which are able to evolve both in the short and long term. This report provides a rational and defines the framework for the RENAISSANCE method.

The report contains two chapters and a bibliography section including main references.

Chapter 1, "introduction", provides the background for understanding the framework. It contains *who*, *why*, *what* and *how* sections in order to help readability of the report itself, and two main sections, concerning the identification of the problem addressed by RENAISSANCE, and the approach proposed to solve the problem. Legacy software systems with their dilemma about what-to-do are identified as the major problem within the software evolution domain, and the reengineering perspective, seen as the systematic transformation of the existing into a new form, is identified as a promising technique to answer the what-to-do question. Based on this background, the RENAISSANCE approach is defined as a reengineering approach which borrows ideas and concepts from Business Process Reengineering state-of-the-art. Objectives and constraints of the method, and benefits of adopting it are pointed out and discussed. Acknowledgements conclude the chapter.

Chapter 2, "the renaissance framework", provides a comparison between the current approach to application management, the maintenance approach, and the proposed one, the evolutionary approach. The chapter suggests the way to introduce and adopt this new technology into company's organization, and defines a framework in terms of a domain, a paradigm, and an abstract model containing generic activities. A two-phase process is proposed to support the overall reengineering activity. The *what-to-do* phase evaluates the existing legacy system combined with the business goals and, as a result, identifies the best re-engineering strategy to implement. Then the *how-to-do* phase supports the implementation of the planned transformation. The remainder of the chapter details all the identified activities. The RENAISSANCE method, defined in the corresponding report, is thus an instance of this framework.

Table of Contents

1 Introduction	1
1.1 Introduction.....	2
1.1.1 WHO is this report aimed at.....	2
1.1.2 WHY to read this report.....	2
1.1.3 WHAT is covered in this report.....	2
1.1.4 HOW does this report tackle the problem.....	3
1.2 The Software Evolution Problem.....	3
1.2.1 Legacy Software Systems	3
1.2.2 The Legacy Dilemma.....	4
1.2.3 The Reengineering perspective.....	5
1.2.4 Reengineering: a <i>What To Do</i> and <i>How To Do</i> activity.....	7
1.3 The RENAISSANCE approach.....	8
1.3.1 Overview	8
1.3.2 Evolutionary Systems.....	9
1.3.3 The Business Process Reengineering (BPR) perspective - An overview	10
1.3.4 RENAISSANCE Objective and Constraints	11
1.3.5 RENAISSANCE Benefits	12
1.3.6 Acknowledgements.....	13
1.4 Outline of the Report	14
2 The RENAISSANCE Framework.....	15
2.1 Introduction.....	16
2.2 Maintained vs. Evolutionary applications.....	17
2.3 The RENAISSANCE Rational, Domain and Paradigm	21
2.4 The RENAISSANCE Abstract Model.....	22
2.4.1 The overall model.....	22
2.4.2 Summary of RENAISSANCE objectives	38
3 Bibliography.....	39

1 Introduction

Contents

- 1.1 Introduction
- 1.2 The Software Evolution Problem
- 1.3 The RENAISSANCE Approach
- 1.4 Outline of the Report

Summary

This chapter introduces the definition of the RENAISSANCE framework. The general introduction defines who is this report aimed at, why to read it, what is covered, and how does the report tackle the problem. Topics covered in the remainder of this chapter include an overview on the software evolution problem with the presentation of the legacy systems dilemma and Reengineering as a promising approach to solve this problem, an overview on Business Process Reengineering from which RENAISSANCE borrows concepts and ideas, and the definition of the RENAISSANCE approach to reengineering legacy software systems, with emphasis on objectives, constraints and benefits of using this method.

1.1 Introduction

1.1.1 WHO is this report aimed at

The intended audience are

- people interested in having an overview on the Software Evolution problem, and focusing on the Reengineering approach;
- senior managers and project managers which are going to start a reengineering project;
- people, both technical and managerial, interested in learning both fundamentals and rationale of the RENAISSANCE method for reengineering legacy systems.

A background in Software Engineering is recommended, even if not mandatory.

Keywords: *application management, software maintenance, software evolution, legacy systems and legacy dilemma, business process reengineering, software reengineering,*

1.1.2 WHY to read this report

The objectives of this report are

- to provide a common background to all the people involved in a reengineering project using the RENAISSANCE method;
- to provide a rationale for the RENAISSANCE method;
- to define a unifying structure for the full spectrum of activities that constitute a project for evolving a legacy system according to the RENAISSANCE method, in the sense that the RENAISSANCE method is an instance of the framework defined in this report.

1.1.3 WHAT is covered in this report

This report covers the following issues:

- an overview on the Software Evolution problem
- an overview on legacy systems reengineering
- an overview on business process reengineering
- objectives, constraints and benefits of using the RENAISSANCE method;
- the definition of the domain of applicability and paradigm of the RENAISSANCE method;
- the definition of a framework for the RENAISSANCE method;

-
- the description of the activities involved in the RENAISSANCE framework.

1.1.4 HOW does this report tackle the problem

A top-down approach has been used in writing this report:

- first, a general introduction to the problem of evolving software is presented: legacy software systems are introduced and Reengineering is presented as a possible and promising approach to the legacy dilemma;
- then, the RENAISSANCE approach to reengineering legacy systems is presented from a general point of view: the rationale of the method, the objectives, and also the constraints and the benefits are extensively treated;
- finally, the RENAISSANCE framework is defined, using a top-down approach again: first, the abstract model is presented from a general point of view; then the main activities are identified, and all of them are detailed.

1.2 The Software Evolution Problem

1.2.1 Legacy Software Systems

Legacy software systems may be defined informally as “large software systems that we do not know how to cope with but that are vital to our organization”. A typical legacy system might be written in assembly or an early-version of a third generation language (such as Fortran 77 or Cobol). It was probably developed using state-of-the-art programming techniques or even software engineering principles; but inevitably software, especially business-critical software, needs to evolve. As early observed by [LEHM 80], evolving software requires remedial action, or its structure will tend to degrade. For the great majority of legacy software, such remedial action has never been taken. So, owing to the maintenance approach to the evolution of software systems, whatever structure originally existed has long since disappeared [BENN 95] . Software maintenance, as defined by ANSI/IEEE-STD-729-1983, is *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment*. This is very simple to define, but extremely difficult to implement efficiently. In fact, without current, up-to-dated, documentation on the continuously changing structure, maintenance is done using the source code, because it is the only reliable source of information about the system. Over time, the software becomes very difficult to maintain, yet the organization’s requests for maintenance become more frequent and more insistent. Statistics confirm this negative trend:

- the top 10 most expensive software errors are maintenance errors [MCCL 90]

- most software lifecycle costs are due to maintenance [MOAD 90]
- companies can not count on having enough trained maintenance personnel [BUSH 90]

These numbers emphasise the need for improved maintenance efficiency and reliability, since software systems, and their maintenance, have several inherent problems, which can be summarized as follows:

- legacy systems are usually complex, unstructured, loosely coupled, or non-cohesive;
- maintenance can create unpredictable ripple effects;
- the documentation is often missing or unreliable;
- applications were written for obsolete software/hardware platform;
- organizations can't keep experienced maintenance programmers.

This leads to legacy systems that lack the ability to evolve to meet ever changing demands in a cost-effective manner.

Moreover, many legacy systems are performing a crucial work for their organization, and usually they represent years of accumulated experience and knowledge. Hence the decision on how to manage them is crucial. Sometimes it could be more convenient to replace the legacy software with a new developed one; sometimes it could be more convenient to restructure it, and sometimes it could be even better to do nothing at all. The decision depends on a mixture of factors, both technical and economic.

1.2.2 The Legacy Dilemma

The issue of legacy systems has become recognized only in the last few years, perhaps because the maintenance costs and applications backlog have increased

As a result of initial technical choices and constraints (i.e. small main storage, efficiency instead of clarity, ad hoc solutions, etc. etc.) legacy systems are typically very difficult to understand, and program understanding becomes a major maintenance activity, and they usually are also large, often monolithic, typically comprising hundreds of thousands of lines of source code, or even more. These technical problems are accompanied by more general management problems. It is more attractive to work on new systems development instead of maintaining old, obsolescent systems; and the necessary skills may be in short supply.

There may be user resistance to change, as well. Despite weak technical foundations, legacy software may be very reliable and responsive to customer needs, and those customers may be heavy users of undocumented features. In the short term, a replacement system may be less reliable and require its customers to do a lot of relearning and rework.

Thus there is a dilemma [BENN 95]. On the one hand, the system is very valuable, and simply replacing it, which is usually the desired solution,

may be too expensive to contemplate because of the huge volumes of on-line data that must be converted, among other reason. On the other hand, the system is becoming too expensive to maintain and the demands of the marketing department for alterations cannot be sustained. And business opportunities might be lost.

Thus, what's the best way to cope with a legacy system? Again, the decision depends on a mixture of factors, both technical and economic: we must trade-off the cost of continuing to cope with the legacy system against the investment needed to improve it and the benefit of easier subsequent maintenance.

Until recently, researchers have largely ignored the problems of legacy systems, preferring instead to study front-end problems. And solution strategies were crystallizing to two main opposite directions: discarding or redeveloping the system. But alternatives to these approaches are now being explored, and results are being applied to industrial pilot projects. Much of this research effort is being expended on an approach named *reengineering*. As detailed in the next section, it covers a wide range of techniques, from simple control restructuring to design and specification recovery in preparation to new forward engineering, to component encapsulation as well.

1.2.3 The Reengineering perspective

Until recently, software engineering concentrated almost exclusively on the definition and improvement of the software development process [STSC 93]. And this produced a lot of important results, ranging from structured analysis, to object oriented analysis, domain and component analysis, CASE environments, etc., which are very useful in developing new systems and forward engineering existing systems well maintained and documented. But legacy systems, created prior these methodologies and tools, usually are very poor in documentation and suffer of years of personnel change and ad-hoc maintenance interventions. This is where reengineering steps in.

According to [TILL 95], reengineering is the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer. This definition emphasises the focus that reengineering puts on improving existing systems with a greater return of investment (ROI) than could be obtained through a new development.

Reengineering is closely related to traditional maintenance, as defined by ANSI-IEEE: maintenance entails making corrective, perfective, and adaptive changes to software, while development focuses on implementing new capabilities, adding functionalities, or making substantial improvements typically by using new computer resources and incorporating new software technologies; reengineering spans the gap between these two activities and exhibits characteristics of both. It provides a mechanism for putting high software maintenance costs under control, recovering existing software assets, and establishing a base for future software evolution; it allows to achieve many goals in software maintenance and to plan for change in existing systems [ARN 93].

Once an organization has defined its maintenance process, software reengineering tools provide support for the migration of legacy software systems into the new maintenance environment; in particular, [STSC 95], they provide support

- to capture design information
- to comply with standards
- to restructure the system, even if unstructured
- to retarget and resource the code
- to support and encourage reuse
- to assess the system
- to elicit business information embedded in the application, and to use them in performing a more general reengineering of the overall company's business process.

The term reengineering is quite broad, and covers a wide variety of techniques. While more details on reengineering and specific techniques can be found in the RENAISSANCE Technical Reports, in the following we cite the [STSC 95] taxonomy of reengineering strategies (with the obviously meaning), just to give an idea of the spectrum of technologies involved in a reengineering project:

- Data Name Rationalization
- Data Reengineering
- Forward Engineering
- Redocumentation
- Reformatting Tools
- Restructuring
- Retargeting
- Reverse Engineering and Program Understanding
- Source Code Translation

Some of these specific techniques will be fully supported by the RENAISSANCE method, while for others references to main knowledge sources will be provided.

Main benefits from reengineering can be summarized as follows:

- **Lower costs** - there is evidence from a number of US projects that reengineering an existing system costs significantly less than new system development. The figures quoted by Ulrich ('The evolutionary growth of software reengineering' in R.S. Arnold, Software engineering, IEEE Press) showed that reengineering a system cost \$12 million compared to an estimated \$50million cost of redevelopment.

-
- **Lower risks** - incremental reengineering of a system means that the risks of each improvement are relatively low. It is less likely that the business will be faced with a system which does not meet its real needs.
 - **Better use of existing staff** - existing staff expertise can be used and staff can develop their skills as the system is reengineered. There is less need to bring in new staff from outside the company.
 - **Revelation of business rules** - as a system is reengineered, business rules which are embedded in the software may become clear. This is particularly likely when these rules relate to exceptional situations.
 - **Incremental development** - reengineering can be carried out in stages as budget and staff are available. The organisation always has a working system available. End-users of the system have time to adapt to system changes and are not faced with a completely new system.

Even if the above discussion provides only an overview on reengineering, it is quite evident that reengineering seems to be a valuable, promising approach for revamping (in the broader sense) existing legacy systems. But this is not enough. Once we have recovered the current situation, we need to look ahead, and we must avoid that both newly developed systems and forward-engineering of reengineered systems become tomorrow's legacy systems.

Owing to the fact that legacy software is basically the result of management inaction rather than a technical deficiency [LEHM 80], we must learn to regard software evolution as an integral part of the development process, not an adjunct. And reengineering must become a continuing part of the process of software development and evolution.

The RENAISSANCE approach we are going to present in the next chapter provides a solution to this more general problem of both migrating existing legacy systems toward evolvable systems, and to build new evolutionary systems.

1.2.4 **Reengineering: a *What To Do* and *How To Do* activity**

Based on the fact that software systems evolve during and after development, but requirements for evolution is hardly ever addressed at the start, Application Evolution aims at providing software built specifically to facilitate evolution.

The major stumbling block on the road to evolutionary systems is represented by the existing base of legacy systems. Evolving over a number of years, legacy systems embody substantial corporate knowledge, including requirements, design decisions, and business rules. In order to effectively use these assets, it is important to develop a systematic strategy for the continued evolution of currently fielded systems to meet changing missions, technology and user needs. However, knowledge of the business rules and technical decisions is often embedded in the code, and it is difficult (often impossible) to recover, owing to many

years of operation, changes, and even personnel change. This leads to systems which do not have the ability to evolve to meet ever-changing demands in a cost effective manner.

But, from this perspective, the existing base of legacy systems is only a today's problem: we have to manage it in a special way, also developing ad-hoc approaches, like reengineering techniques. As already said, what is really imperative is a paradigm shift from today's develop-then-maintain systems to systems that continuously evolve, in order to start by scratch with new systems and also recovered legacy systems, that is systems ready to be managed with evolutionary concepts.

Reengineering offers an approach to migrating a legacy system towards an evolvable system.

But the impact and extent of change caused by reengineering can be very extensive, and they can have profound implications on the overall organization of the company.

From this point of view, reengineering is not always the best solution for a legacy candidate; and even if it is, different strategies may be applicable, and all of them should be carefully evaluated in order to contain the effort and speed up the process.

This means that the overall roadmap for managing application evolution must contain a specific activity concerning reengineering. Owing to the complexity and specificity of this task, RENAISSANCE prefers to split the overall reengineering activity in two main phases: the first one for the assessment of applications and the decision of the most appropriate evolution strategy, and the second one for the transformation subtask. The RENAISSANCE framework reflects this distinction.

There are some fundamental questions to be answered before starting a reengineering project, that is before moving throughout the reengineering roadmap:

- Is reengineering the right strategy?
- Which is the best reengineering strategy for the considered application?
- What is the cost/benefit ratio of reengineering, and what is the cost/benefit ratio of not reengineering at all?

The future of the candidate system depends on the answers to these critical *what to do* question. A formal assessment of an existing application to determine whether to maintain, reengineer (with which strategy), or retire that system is necessary. The RENAISSANCE *What To Do* phase will address these questions, while the *How To Do* phase will provide guidelines for implementing the planned transformation.

1.3 The RENAISSANCE approach

1.3.1 Overview

RENAISSANCE supports the transformation of a legacy software system to an evolutionary software system, that is a system which is able to evolve both in the short and long term.

An overview on evolutionary systems and on business process reengineering, from which this method borrows concepts and ideas, are presented as necessary background for the method.

Then, objectives, constraints and benefits of adopting RENAISSANCE are discussed.

Acknowledgements conclude this chapter.

1.3.2 Evolutionary Systems

The concept of evolutionary systems represents a break with the old “develop then maintain” model of software in favour of a new lifecycle model that features continuous evolution. Evolutionary systems are defined as systems that are capable of accommodating changes over an extended operational life; this changed lifecycle is quickly becoming an imperative for software organizations.

The technology that supports evolutionary systems must enable

- the creation of systems that are designed to evolve, and
- the transformation of today’s legacy systems into evolvable systems.

The need for software evolution is most evident in systems built and maintained in the context of continuously evolving requirements, rapid changes in technology, and an incremental process of understanding real needs.

[TILL 95] recognizes four critical needs for evolutionary systems:

- the need to develop the right software, which originates with the problem of requirements specification;
- the need to develop the software right, which refers to the problem of using good practices to develop a well-engineered software;
- the need to develop the software rapidly and affordably, which refers to economics of software development;
- the need to build systems which evolve together user needs.

While we can suppose that new projects will be started with these needs in mind, for legacy systems there is still a need for evolutionary principles to be applied. There needs to be evolutionary technology applied to existing systems so that they will be nearly as easy to maintain as newly developed systems. But this will not be accomplished without some reengineering.: there will be need to refresh, rewrite, redesign, rearchitect, and redocument existing legacy systems. And quite often, in order to achieve real benefits, it is not sufficient to perform the technical reengineering of a legacy application, but the complete business systems, whose logic is often embedded in the legacy application itself, must be rethought and reengineered. This Business Process Reengineering perspective is detailed in the next section.

1.3.3 The Business Process Reengineering (BPR) perspective - An overview

To discuss the legacy issue is to raise fundamental questions about how the information systems organization optimizes the use of information technology in the business. Instead of viewing legacy applications in purely technological terms, managers must look at processes that generated legacy applications in such a way as to make them problematic. This shift in perspective requires that legacy issues must be understood within a wider business context, called Business Process Reengineering (BPR) [CSC 96].

BPR is a product of the 1990s. Its exact origin is the subject of some dispute, but most attribute the popularization of the term to [HAMM 86], which brought the concept into fashion; then [HAMM 93] started a management revolution, and the book has become the manifesto for corporations interested in reinventing themselves in the post-industrial business age.

One of the fundamental precepts of reengineering is that the status quo is unacceptable; that the current system cannot be sufficiently enhanced to achieve the necessary dramatic reduction in cost, increases in productivity, and cuts in time-to-market. To satisfy these objectives is the aim of BPR, defined by [HAMM 93], as “the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service, and speed”.

The reader should never confuse BPR with software reengineering; we have included this section only to present a general view on reengineering, and to discuss how software reengineering is being used to support the overall business process reengineering. This issue is discussed in the remaining of this section.

Legacy software was developed to support business functions within the traditional organizational structure. Thus, we are left with systems that are marketing-oriented, or manufacturing-oriented, etc. etc. Software reengineering provides support to business process reengineering in two ways [STSC 93]:

- Reengineering software captures the design information behind the software. Using tools and techniques now available the design information can be broken up into chunks that are functionally cohesive. These chunks are then re-aggregated, that is analyzed and regrouped around the newly identified business core processes.
- Reengineering software aids in the identification of business rules from legacy source code. In fact, it is quite often the case that embedded within legacy systems are the implied rules that governed how the organization was run at the time the system was developed. By extracting these business rules, an organization can better understand and, later, codify the rules by which they conduct business. Combining these information with the existing and/or recovered design information, components of the legacy software which correspond to business areas could be identified and extracted. These components could then be re-aggregated to represent a new

system that is oriented to support the business processes as defined by business reengineering.

Software reengineering, and, more generally, business process reengineering, is a drastic and dangerous step, but it could be the only way to cut down the amount of money and resources that some companies blatantly waste for no apparent reason other than by way of their inefficient structure or used technology.

Apart technology, looking at case studies, and both success and failure stories, [STSC 93] recognizes as mandatory to consider the so-called human factor, that is:

- Early user involvement. The use of BPR strongly affect a company and the way in which such company does its business; because this is a critical investment, users must be continuously involved in eliciting, assessing, and redefining processes.
- Redesign the corporation as a whole. Main requirements are listed in the following:
 - team workers, grouped by skill and task;
 - workers empowerment, to have more efficiency;
 - internal and external collaboration enforcement, to overcome specific technical problems.

The first step for evolving processes is to assess the existing process. Metrics, like SEI-CMM (Capability Maturity Model by Software Engineering Institute) should be used for this assessment, whose result can be used to drive the following improvement.

RENAISSANCE borrows from BPR the need to rethink (or reengineer) the entire business process, when it is a legacy one, but it concentrates only on reengineering information systems. This means that providing support to BPR is clearly out of scope of RENAISSANCE. Nevertheless the method could be used within a more general BPR framework, in order to achieve the above described results: capturing design information and embedded business rules, which are the basis for a success BPR-story. This is the rationale for having presented here a quick overview on business process reengineering.

1.3.4 RENAISSANCE Objective and Constraints

The main objective of the RENAISSANCE project is therefore:

To develop a *systematic method* for system evolution and re-engineering which is geared to the requirements of the commercial systems domain.

The RENAISSANCE method will take into account technology changes in this domain which have led to customer pressure to migrate applications from centralised mainframes to object-based, distributed client-server systems. It will include support for the specific requirements of 3GL and 4GL system evolution.

The key sub-objectives of the project are therefore:

Error! Style not defined.

- To provide techniques for modelling the principal structures of 3GL and 4GL business applications and to give advice on the use of architectural modelling as a basis for system understanding, re-engineering and distribution of these systems.
- To provide techniques and technologies for converting centralised legacy systems and associated data to distributed client-server system architectures.
- To provide advice to managers responsible for planning the evolution process on evolution strategies, organisational factors which must be taken into account in the planning process (e.g. process changes, training, etc.) and the risks and economic of applying these strategies in the evolution process.

The principal business objectives of the industrial partners in the RENAISSANCE project are to improve their capability to offer profitable services in the area of system evolution and to increase their return on investment in their software assets. These business objectives will be met by developing a more *methodical* approach to evolution and re-engineering which is consistent with current development and maintenance practices used in industry.

1.3.5 RENAISSANCE Benefits

The principal result of the project will be a ***RENAISSANCE method handbook*** which describes a generic method for software evolution with variants which reflect the specific requirements of 3GL and 4GL evolution. The method handbook, will be widely disseminated by ensuring that it is published by a commercial publisher. The handbook will include information about evolution processes, system models and documentation for evolution, and rules and guidance for applying the method. It will also integrate the results of the project which are embodied as consultancy reports described below.

This handbook will be supported by a set of more detailed ***RENAISSANCE consultancy reports*** and associated training materials which may be used to develop internal and external consultancy services in the areas of system maintenance and evolution.

These reports are:

- System modelling for evolution.
- Migrating to distributed client-server systems.
- System modelling for evolution.

A list of expected benefits is reported in the following:

Enable Business Process Change

- *Functionality and Flexibility*
- *Speed to market and response time*

-
- *Re-engineer or replace obsolete processes*

Improve Productivity

- *Employees and end-users*
- *Processes*
- *Interfaces to Processes*

Support Corporate Initiatives

- *Customer services and satisfaction*
- *Competitive response to existing and new markets*
- *Re-organisation, downsizing and decentralisation*
- *Alignment of corporate and IT goals*
- *Improve revenues and profits*

Control Costs

Improve Information Access

- *Improved access and usability of corporate data*
- *Improved accuracy of process data*

It must be noted that the motivations that drive RENAISSANCE method adoption fall in two category:

- *enhancement of the market competitiveness,*
- *improvement of the IT performances.*

1.3.6 Acknowledgements

There is a lot of interest about the Reengineering theme, especially on Business Process Reengineering. We have found a huge amount of material on the argument, both in literature and on the web, lots of authors, papers, books, technical notes, and reports, which have helped us in having a manifolded view of problems, needs, and state-of-the-art of this emerging paradigm; lots of knowledge sources which have helped us in defining, organising and developing the RENAISSANCE framework and method. Our work has been particularly influenced by some of these knowledge sources: their ideas and thesis have been collected, integrated in a coherent and global view, developed and further engineered. We cite such sources in the following:

- Imai, M., "Kaizen: The Key to Japan's Competitive Success", McGraw-Hill Publishing Company, New York, 1986

- Hammer, M., “Reengineering Work: Don’t Automate, Obliterate”, Harvard Business Review, 1990
- Hammer, M., Champy J., “Reengineering the Corporation”, Harper Collins, New York, 1993
- Software Technology Support Center (STSC), USA, Technical Report, “Reengineering Technology Report”, Volume I, II, 1993
- AMES Consortium, AMES ESPRIT project #8156, “AMES Methodology and Process Model”, 1994
- Tilley, S., “Perspectives on Legacy Systems Reengineering”, Software Engineering Institute, Carnegie Mellon University, Draft Version 0.3, 1995
- American DoD Technical Report, JLC-HDBK-SRAH, “Software Reengineering Assessment Handbook”, Volume I, II, Version 2.0, 1995
- Ganti, N., Brayman, W., “The transition of Legacy Systems to a Distributed Architecture”, John Wiley & Sons, 1995

1.4 Outline of the Report

Chapter 1 introduces the general problem of software evolution, defines the objectives of the method, and defines both constraints and benefits of using RENAISSANCE .

Chapter 2 defines the RENAISSANCE framework for evolving legacy software systems. It is suggested the way in which the RENAISSANCE technology should be introduced in a company's organization, and a comparison between maintained (the current practice) applications and evolutionary (the desired practice) applications is presented, as rational for the framework. A domain of applicability and a paradigm are defined for the method, and the abstract model is presented and its activities are detailed. A summary of RENAISSANCE objectives ends the chapter.

Chapter 3 contains the bibliography of this report.

2 The RENAISSANCE Framework

Contents

- 2.1 Introduction
- 2.2 Maintained vs. Evolutionary Applications
- 2.3 The RENAISSANCE Rational, Domain and Paradigm
- 2.4 The RENAISSANCE Abstract Model

Summary

This chapter defines the RENAISSANCE framework. Topics covered include a comparison between maintained and evolutionary software applications, the definition of a rational, domain and paradigm for the method, and the detailed definition of the full spectrum of activities which constitute the RENAISSANCE abstract model.

2.1 Introduction

RENAISSANCE aims at those organizations that have to deal with *legacy software applications*, that is software applications with high business value but which are difficult to maintain, providing a method to drive the continuous evolution of such applications both in the short and long term.

Because migrating a software application from the current status of *legacy* application toward a status of *evolutionary* application involves reengineering the application itself, RENAISSANCE proposes a method to support the full spectrum of activities involved in a reengineering project. This method takes advantage from the current state-of-the-art in reengineering: existing approaches and techniques have been analysed, organised in a coherent view, and used as basis for the methodology we have defined. The RENAISSANCE method supports the overall reengineering activity, and provides full coverage on specific reengineering techniques, like the migration from a centralised to a client/server n-tiers architecture.

This chapter defines a framework for the RENAISSANCE method, which can be seen as an instance of this framework; the method is defined in the corresponding report.

Because a reengineering project can have a huge impact on the company's organization, it is necessary to plan the way in which the reengineering approach is taken. In fact, a planned approach is mandatory for a successful transition from a legacy application to an evolutionary one. In general, transitioning effective practices, processes, and technologies consists of a series of activities or events that occur between the time the new idea is encountered and the daily use of that idea.

RENAISSANCE suggests to use the Conner and Patterson's Adoption Curve, [STSC 93], to insert this new technology within an organization.

The suggested adoption curve is depicted in figure a, and it is detailed in the following.

After encountering a new process or technology, potential customers of that technology increase their awareness of its usage, maturity, and applicability. If the process or technology is promising, then customers try to better understand its strengths, weaknesses, costs, and applicability. The first activities in the adoption curve take a significant amount of time. Promising processes and technologies are then evaluated and compared. To reduce risk, customers usually try new processes or technologies on a limited scale through beta tests, case studies, or pilot projects. A customer then adopts processes or technologies that prove effective. Finally, refined processes and technologies become essential parts of an organization's daily process: we name this as institutionalisation.

In adopting the RENAISSANCE technology we recommend to use this proved approach.

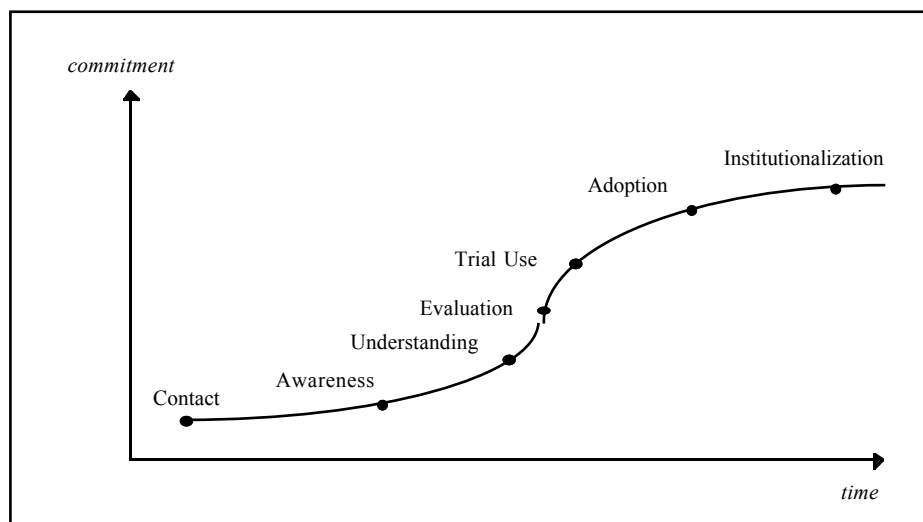


Figure A: Conner and Patterson's Adoption Curve for a new technology

2.2 Maintained vs. Evolutionary applications

Until recently, most *software systems* (simply *systems* in the following, if not differently specified) were developed under the implicit assumption that it would be maintained for a period of time and, at some point, replaced with a new system. In the meantime, the system, if valuable, was *maintained*.

This means that, traditionally, maintenance has been considered a separate phase from development. Frequently, the maintenance team has different skills from the development team, and quite often they use different tools as well. This discontinuity between development and maintenance has led to many problems.

Many organizations are faced with the problem of maintaining aging software systems constructed to run on a variety of hardware, programmed in obsolete languages, using old technologies, and suffering from the detritus that accumulates from years or even decades of maintenance. Most organizations recognize this is becoming a difficult task, particularly as the "old pros" of the maintenance organization move on to other activities, and new recruits need to be trained to address often-unattractive maintenance activities on old systems.

Application software maintenance has been recognized a resource intensive activity, which requires about 50-90% of the total life cycle cost; and this trend of high maintenance costs is continuing to increase, owing to increasing age of software applications.

An organization incurs a huge software maintenance cost mainly because of

- volatile user requirements, and

- deteriorating software maintainability (due to lots of reasons, spanning from company reorganisation or growth, to change in personnel, and so on).

A volatile user environment generates new user requirements. These requirements are translated into maintenance requests which call for modifications or enhancements to the existing system. Hence, the higher the volatility of the user environment, the higher is the maintenance effort and cost; and, as the software system is being modified and enhanced over a long period of time, its maintainability degrades: with frequent interventions, the number of inputs and outputs, the number of functions, and inter-module interactions within the software increase, leading to higher complexity. In addition, these changes are often neither well-integrated into the existing software design, nor well documented. The obvious result is a drastic deterioration of system structure and quality, which leads to increased effort for each maintenance request as the software system ages. The increase in the effort per request exacerbates the total cost of maintenance, and forces companies to shift their focus from software development to software maintenance tasks. Evolving a system to meet changing user needs is reflected by a need to change software during long lifetimes.

The lesson learnt from decades of maintenance is that systems evolve both during and after development, but the requirement for the evolution is hardly ever addressed at the start. This has led to a new perspective, which is becoming broadly accepted: the *evolutionary* perspective on software development. That is: it makes sense to design software for change in the first place, in order to reverse the current trend in large, complex systems.

The evolutionary perspective recognizes that:

- *System lifetimes are very long.* Most currently operative systems were designed to be obsolete long ago. The current budgets cuts and downsizing tend to extend the desired useful lifetime of systems ever further.
- *Requirements change during the system's lifetime.* If the system is of sufficient size and complexity, it is practically certain that requirements change even during the design.
- *The opportunity and/or need for change is time-critic.* For any systems, at least the underlying technology changes rapidly, and they should adapt themselves to such changes to best fit the market and user needs.
- *Systems are often components of other systems.* They must evolve together with the environment.
- *Computational resources are not usually the principal inhibitor for evolution.* Apart few exceptions like physical limitations such as size of memory, the needed evolution can not be accomplished by merely buying increased computational capabilities.

The concept of evolutionary systems represents a break with the old “develop and maintain” model of software production in favour of a new lifecycle model that features continuous evolution. This changed lifecycle

approach is quickly becoming an imperative for software organizations, especially when the targets are systems to be built and maintained in contexts of continuously evolving requirements and rapid changes in technologies.

Evolutionary systems are defined as systems that are capable of accommodating changes over an extended operational life. The technology that supports evolutionary systems must enable

- the creation of systems that are designed to evolve, and
- the transformation of today's legacy systems into evolvable systems.

There are at least four main differences between maintained and evolutionary systems. The objectives of evolutionary systems can be summarized as follows:

1. To replace separate approaches to pre and post-deployment by a single incremental approach to evolving a system; that is: the development of a software system is never intended to end. After early fielding of an initial capability, the system should continue to become more capable over time.
2. To replace point solutions driven by specific system requirements by parameterized families of solutions. The approach is to institute and support software product families that, in turn, are matched against requirements resulting in both changes to the requirements and adaptations of the products.
3. To make use of software asset that already exist instead of starting design from scratch. Examples of such assets are COTS, reusable components, standard architectures, and existing systems.
4. To involve the end user in the software development process in order to achieve the necessary functionality, early user buying, and elimination of false starts. In addition, more control over the software portions of the system are placed directly in the hands of its users.

Summarising, the new paradigm for evolution demands for:

- early user involvement;
- the use of currently available software assets;
- families of solutions (or domain-reusable solutions); and
- the use of a single incremental approach to evolving a system.

The need of evolvability over the long term must be addressed from the very start of new systems development. With evolution as primary objective, the greatest emphasis must be put on *engineering for change*, which addresses

- the need to reduce long-term life cycle costs as opposed to short-term development costs;
- the need to design software with new requirements in mind;
- the need to prepare the software to integrate new functionalities, rather than to patch the software to insert such new functionalities.

The main obstacle in adopting this paradigm is represented by the large base of existing legacy systems, most of them currently performing crucial tasks. These systems have been developed and evolved over the past quarter century with a variety of languages, operating systems, hardware, development methods, tools, and idiosyncrasies; and documentation is often poor or even unexistent. However, because of the existing investment in these applications, the critical roles they perform, and the expense of (re)developing them from scratch, it is necessary to protect the existing asset. This means that it is unfeasible to replace these legacy systems with redeveloped ones, built on top of the evolutionary paradigm; it is necessary to turn legacy, maintained systems into evolutionary ones. There needs to be evolutionary technologies applied to existing legacy systems so that they will be nearly as easy to maintain as newly developed systems. This will not be accomplished without some *reengineering*. There will be need to refresh, rewrite, redesign, and redocument. However, over the long term, these costs can be more than justified by the savings achieved in maintaining otherwise-unmaintainable systems. Which refreshing strategy the system needs is the main problem we have to face out. The RENAISSANCE method focuses on this problem, and it gives guidelines to ensure evolvability.

Building and evolving large software systems involves very large upfront investments in time and effort for learning and relearning requirements from scratch and for validating results. Concerning the software engineering domain, much of this investment is precluded by the use of architectural and product models as well as tested and compatible components. The three enabling keys to implement the evolutionary concept within the software domain are:

- *Modelling and Architecture*. These areas address global consistency and exploitation of commonality in large software systems. The term Modelling is most often applied to the discovery of operational concepts and the specification of requirements for the software. Software Architecture is a type of modelling that is more often applied to the design phase of software development. Both of them deal with creating abstractions that help in understanding.
- *Information-rich systems*. The collection, control, traceability, and utilization of voluminous information about the system and its history is fundamental to prescribe the right intervention on the system itself.
- *Predictability*. Enabling impact analysis and change simulation on the system increases the confidence in applying a change.

Changing the paradigm from *maintenance* to *evolution* is not an easy task, and it is necessary both understanding and transforming the legacy asset. The remaining sections detail the RENAISSANCE framework, and provide

- a definition of a domain of applicability and a paradigm , as well as a rational; and

- a unifying structure for describing all the activities involved in a reengineering project; that is an abstract model for classifying and organizing method activities.

2.3 The RENAISSANCE Rational, Domain and Paradigm

The domain of applicability of the method are legacy software systems, and the focus is on the concept of evolution, which can be defined as the attitude to respond and adapt to changes. In fact, the maintenance approach implicitly assumes static requirements, while the evolutionary one recognizes the current trend: volatile requirements for better fitting the market and exploit new, continuously changing, opportunities;

RENAISSANCE borrows and uses concepts, ideas and experience from Business Processes Management, and apply them to software evolution. Two approaches are explicitly borrowed: BPR and Kaizen, which represent the opposite borders of the process management continuum.

BPR is associated to rethinking a process from scratch. But, because it seeks radical change through designing whole new processes, it is not sufficient to guarantee evolutionary processes, and it can be seen just as a recovery strategy. Instead Kaizen strategy, as defined by the Japanese leading consultant Masaaki Igarashi in his international best-seller named '*Kaizen*', demands for continuous improvements without dramatic changes, that is ongoing slow-tuning of the process. table a summarizes both the approaches.

Business Process Reengineering	Kaizen - Continuous Improvement
Radical Redesign and creation of new processes	Continuous improvements to existing processes
Broad, organization-wide	Single teams or functions
Destroy the old, and begin fresh	Standardize and stabilize existing processes
Top-down	Bottom-up
Major structural changes force new behaviour	Training and culture change drive new behaviours
High investment and risk with little room for error	Moderate investment and risk by learning as you go

Table A: A comparison of BPR and Kaizen approaches

From the RENAISSANCE perspective, which focuses on the evolution of software systems, the two following considerations are important:

- continuous improvement of existing systems is needed, but,
- owing to the current practice of most legacy systems, it is generally hard, expensive and even unrealistic, to satisfy expectations (i.e. save time and money) in a reasonable time, simply tuning system elements.

The simple solution adopted by RENAISSANCE consists in balancing the BPR and Kaizen approaches, in the same way software engineering principles suggest to balance top-down and bottom-up approaches during software systems development.

RENAISSANCE proposes the following definition as a paradigm for the evolution of software systems:

1. A stable basis is recovered from the existing software system through a reengineering approach.
2. Kaizen is adopted on top of the recovered system to continuously tune the system to the changing environment in an evolutionary-fashion.

Reengineering, by itself, is not the total solution to the problem of evolving legacy software systems, but it is a bridge to such a solution, if it prepares the system to evolve in the long term (i.e. by building or transforming the architecture). The RENAISSANCE method provides support to this paradigm.

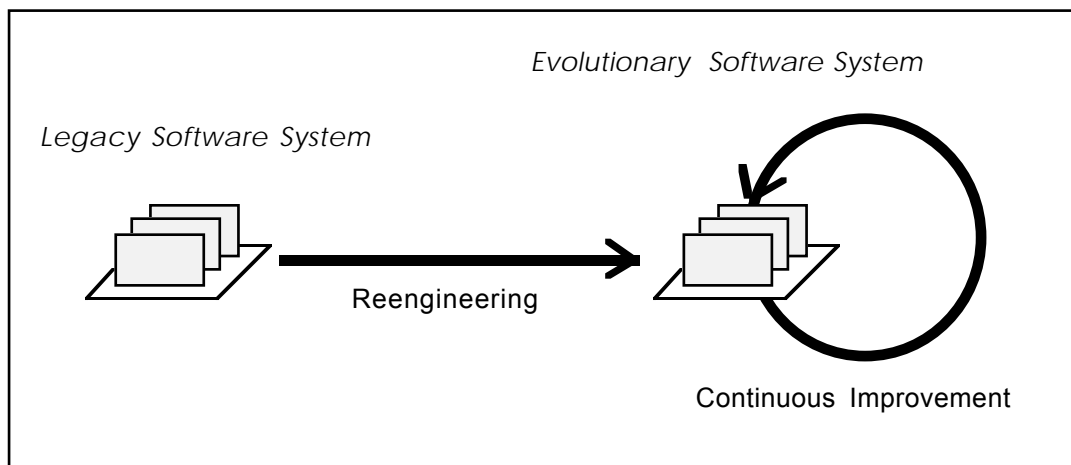


Figure B : The RENAISSANCE domain and paradigm

2.4 The RENAISSANCE Abstract Model

2.4.1 The overall model

This section defines the RENAISSANCE process framework (or abstract model) as a unifying structure for the full spectrum of activities that constitute a project for migrating a legacy system toward its evolutionary dimension.

In the remainder of this discussion, we name such a process of migration simply 'reengineering' or 'reengineering project'. This section can hence be seen as a model for planning a reengineering project, and for identifying,

defining, and assessing both technical, managerial, and other project related factors that characterize the reengineering effort.

With the rational provided by [AMES 94], [SRAH 95] and [TILL 95], RENAISSANCE considers a legacy software system as a collection of *subsystems*, and uses *view* and *role* concepts as tools to provide a coarse grain representation of the system itself:

- subsystems representation is used to break down the complexity of the system and to reduce the effort of reengineering;
- views can be seen as filters on the process; intuitively, there are relationships from one view to the others, because the overall system is the amalgama of the views, and not just the concatenation of them. The amalgama is done through activities involving roles
- roles can be seen as tags for process agents, whose activities can cross views.

RENAISSANCE defines three intuitive views and three intuitive role categories, as depicted in figure c.

The three views are:

- Technical view.

This view represents the technical perspective of the system; i.e. it contains the technical knowledge of the system and the development and maintenance processes, or, in other words, the software, the documentation, the operational supporting environment, the core technologies, etc. etc. are visible through this view.

- Economic view.

This view represents the economic perspective of the system. Economic models, business value of the system, cost estimations procedures, etc. and the process to apply and use them are visible within this view. The focus of this view is on *understanding business values* and *preparing options for decision making*.

- Managerial view.

This view represents the managerial perspective of the subject system, which is a mixture of technical, economic and other issues. It focuses on *decision making*.

The three general categories proposed by RENAISSANCE to group roles are described in the following, while (specific) roles belonging to the identified categories will be detailed in the method definition, because they are not relevant for the framework:

- Strategic role category.

The objectives of this role category comprise:

- identification of system's needs
- continuously striving for quality improvement
- managing organization resources efficiently

- reducing costs where possible while maintaining quality standards
- defining marketing strategies

Agents which belong to this role category are primarily involved in activities of the Managerial and Economic views.

Strategic management focuses on:

- long-term
 - multiple projects
 - resource management
 - budget trade-offs
 - knowledge build up
- Operational role category.

The objectives of this role category include:

- to provide effective project control
- to negotiate contract with customers
- to comply with organisational strategy

Agents which belong to this role category are primarily involved in activities of the Technical and Economic views.

Operational management focuses on:

- short to medium term
 - single project with fixed budget
 - decisions within the framework of a given assignment/contract
- Service role category.

The main objective of this role category comprises:

- to manage and provide services to satisfy objectives of the other two role categories.

Agents which belong to this role category are primarily involved in activities of the Technical view.

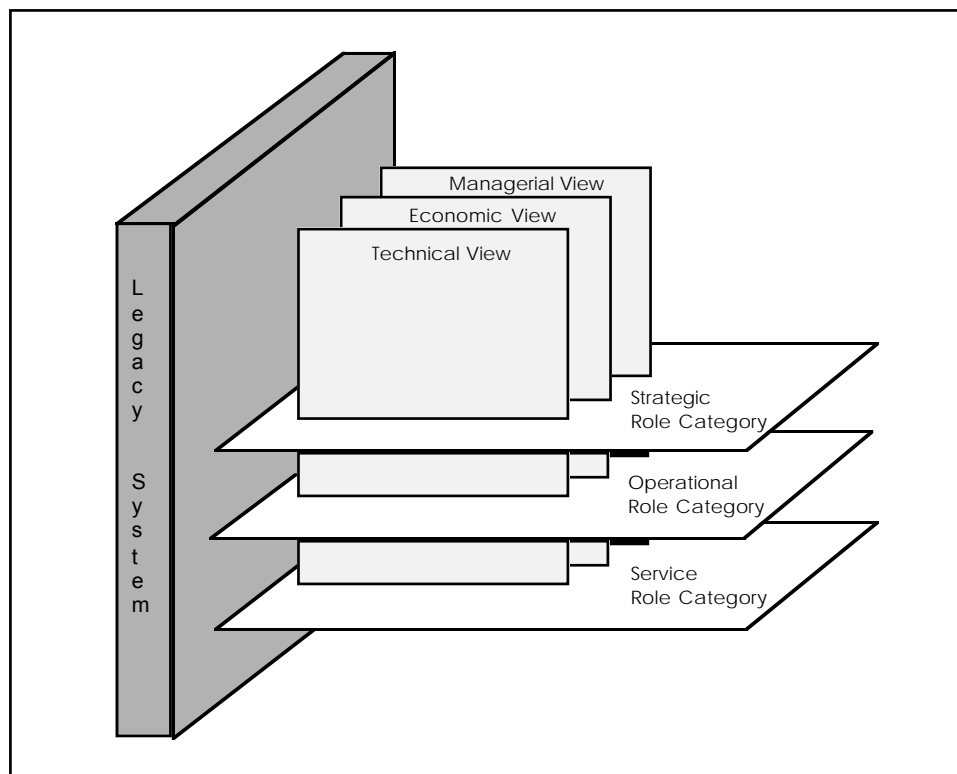


Figure C: The RENAISSANCE three views and role categories of a system

Role categories, whose objectives and main activities can be summarized as in table b, are used to provide a structured view on a system; details can be found in [AMES 94].

Role Category	Objectives are to	Activities include
Strategic	identify the need for maintenance	business case definition risk analysis and assessment product portfolio management
	continuously strive for quality improvements	quality assurance quality reviews for the service, operating procedures, products and customer satisfaction
	reduce costs while maintaining quality	monitor resources review resource usage mandate improvements
	manage organizations resources efficiently	define resource and project management procedures training
	define a marketing strategy	marketing

Operational	effective project control	monitor progress against plan considering risk and resource management and priority planning
	negotiate contracts with customer	produce a maintenance plan for the customer produce a cost breakdown for the service provision take into account the legal aspects of the service provision project a good corporate image
	promote use of maintenance service	marketing of services including technical demonstrations, seminars
	smooth handover on completion of contract	documentation management configuration management release control management closure of intervention
Service	provide appropriate application management services	application understanding design recovery application analysis problem analysis localisation of the problem solution analysis impact analysis solution specification code modification regression testing testing updates update documentation updating operating procedures acceptance testing re-insertion quality assurance review official approval of changes closure of intervention

Table B: Role Categories Objectives and Main Activities within the Software Domain

Based on the previous structure, the RENAISSANCE reengineering process framework is shown in figure d and detailed in the reminder of this section.

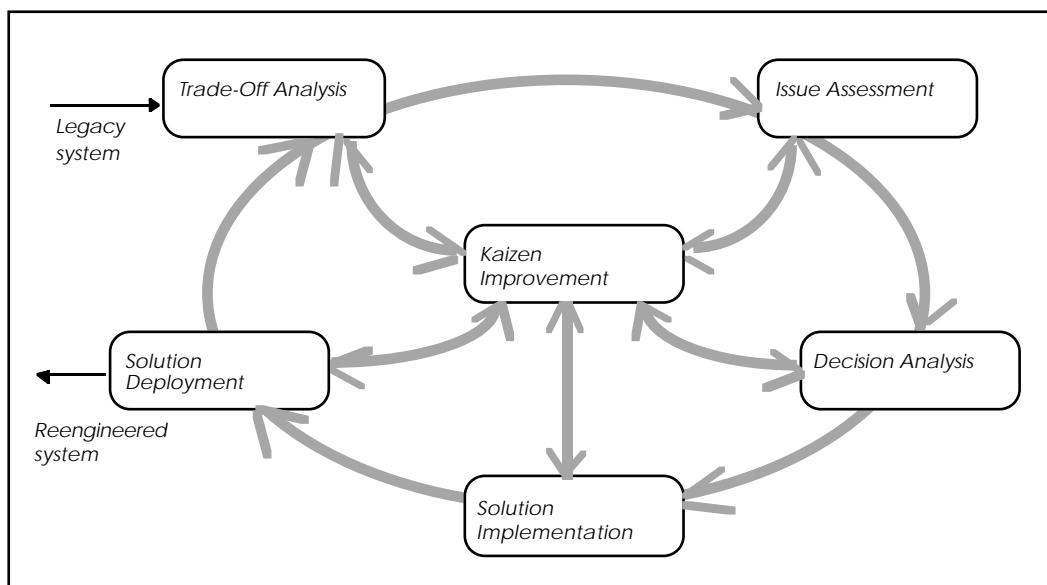


Figure D : The RENAISSANCE reengineering framework

The framework encompasses the following phases, which are detailed in the remaining sections:

- Trade-Off Analysis.

A trade-off analysis is the starting point for each reengineering effort. Current system, open technical issues, technical market trend and business goals must be deeply investigated in order to realize what is called “*pre-planned product improvement*” by [TILL 95], that is to design the software for future changes.

The aim of this activity is to analyse possible changes, especially technologies trade-offs, which could be even largely independent from the quality of the system itself; i.e., the demand for client/server paradigm could be mandatory even if the current system is a reliable, well-documented and structured huge centralised executable.

- Issue Assessment.

The scope and direction of the reengineering effort are established in this phase, through analysis of the legacy software, which is assessed with long-term target in mind.

This phase primarily aims at assessing the current status of the application (i.e. in terms of technical and environmental issues, as well as business issues) in order to decide if it needs to be reengineered. The current status is combined with the result of the trade-off analysis to choose the right long-term strategy. The final result is a scenario of possible directions for conducting the reengineering activity.

- Decision Analysis.

This is a crucial phase, and it represents the concentration point for the flow of information coming from the other activities. The best

reengineering strategy and the plan must be decided on the basis of the variety of information produced by previous phases.

- Solution Implementation.

On the basis of the accepted reengineering plan, a solution must be designed with the evolution concept in mind. Possibly, the solution should solve the family of problems pointed out by the analysis, and the use of state-of-the-art solution patterns should be strongly encouraged. The new solution is then implemented.

- Solution Deployment.

The new implemented system is validated and, if accepted, deployed.

- Kaizen Improvement.

Everything must be continuously refined. This means that:

- the reengineered software system has to be continuously refined;
- the way in which the analysis is done, the decision is taken, the solution is implemented and the new system is deployed has to be continuously revised and refined. In other words, the method itself, defined on the basis of this framework, has to evolve.

It has to be noted that the integration among activities needs a repository (we call it *System Repository*), which contains all the information concerning the system under reengineering, ranging from assessment results to candidate strategies report, software manifests, and so on. The System Repository supports the cooperation among activities for a subject legacy system; it allows incremental building, and exchange of information among roles involved in the activities.

The arrow from Solution Deployment to Trade Off Analysis models both the incremental approach adopted by RENAISSANCE in reengineering systems (subsystems identification, reengineering and integration, as it will be detailed in the method) and the new process in case of change of the renewed system. In both the cases, the process takes advantage of the existing System Repository for that system, in order to reduce the effort.

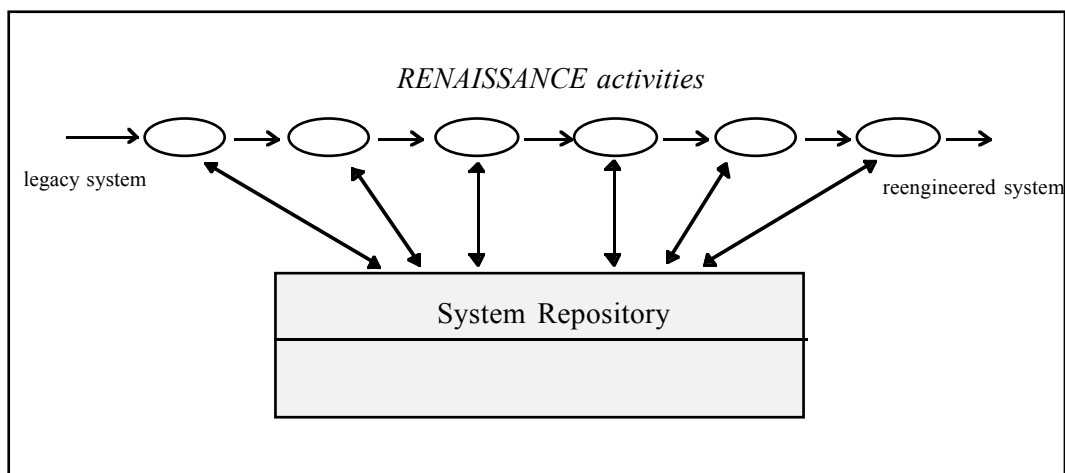


Figure E : Activities Cooperation via a System Repository

The above introduced phases, which constitute a process framework for reengineering, are described below in terms of the generic activities that are necessary to fulfil their intended function and of the roles of agents which have to perform the activities. The RENAISSANCE method, as defined in the corresponding report, is an instance of this framework both in terms of activities and roles per task.

There is an emphasis put by the six-phases framework on the initial evolution decision assessment, and on the concept of continuous improvement derived by the Kaizen approach. Half of the process is on the decision of “*what to do*” with the current system; the second half of the process implements the adopted strategy and answers the “*how to do*” question; the Kaizen phase models the continuous thinking of the process (and the target system) with the aim of refining it (figure f).

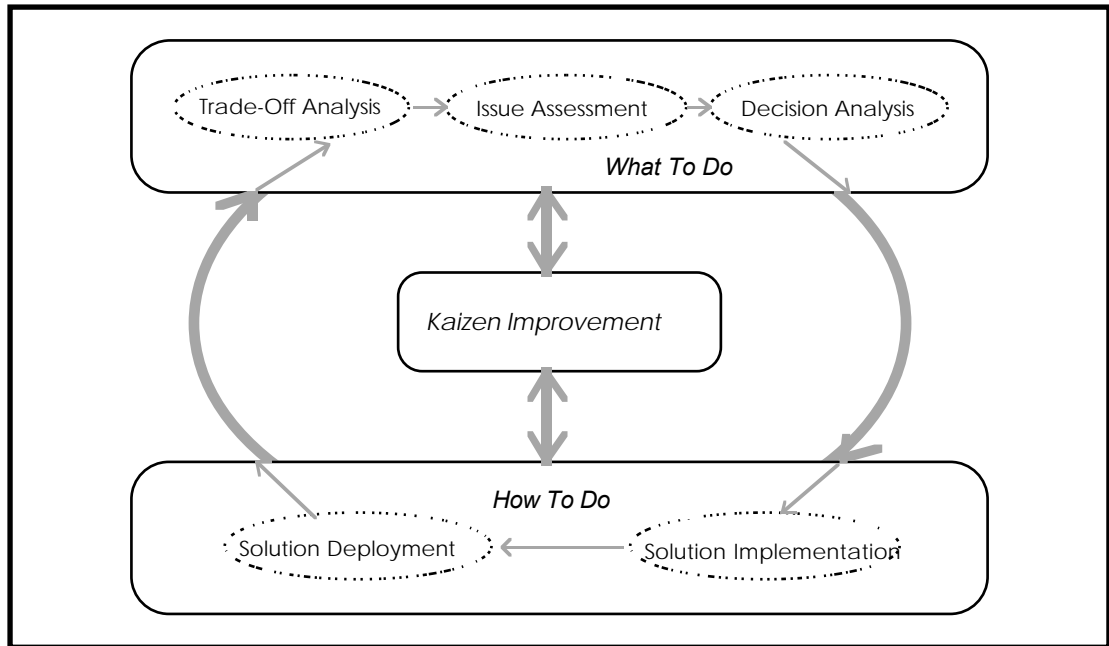


Figure F : The logical view of the RENAISSANCE framework

RENAISSANCE puts a strong emphasis on the evolution decision process (the *what to do* process) . This emphasis will also be reflected by the structure of the RENAISSANCE method. There will be two distinct phases: the first one will define the method for assessing an application and taking the most appropriate evolution approach, and the second one will define the method for implementing such an approach.

Next section defines which evolution approaches are supported by RENAISSANCE, and the reminder details the activities of the framework.

2.4.1.1 Evolution Strategies

In the context of evolution projects, the choice of the right evolution strategy plays a fundamental role in the overall process. The right strategy to be adopted in evolving a system depends on several factors, including the existing legacy asset and the desired target system. RENAISSANCE formalizes this with the *what to do* phase of the framework, which concerns the evaluation of existing and target systems.

It has to be noted that even if the RENAISSANCE method is intended to be a general method of wide applicability, techniques required by the pilot applications that are going to be used for the assessment of the method itself are analysed in more details. The method focuses on a subset of the possible evolution strategies.

The following table summarizes the evolution strategies which will be supported by the method; details can be found in the RENAISSANCE documentation (reports D5.1D “Baseline Review Report”, and D3.3 “Evolution Strategies”).

Strategy	Characteristics
Maintenance	Bug-Fixing, adaptations, etc. without changing the structure or architecture.
Re-vamp	The user interface of a system will be updated to a more advanced technology, i.e. character mode to GUI. The general structure of the software will not be changed.
Re-structure	The structure of the software will be changed. The underlying hardware will not be changed.
Re-architecture	The structure of the system will be changed as well as the underlying hardware.
Re-design with re-use	A new system will be created integrating re-usable assets of the existing system.
Re-placement	The existing system will be replaced by a new developed system without regarding the existing one.

Table C : Evolution Strategies supported by RENAISSANCE

2.4.1.2 Trade-Off Analysis

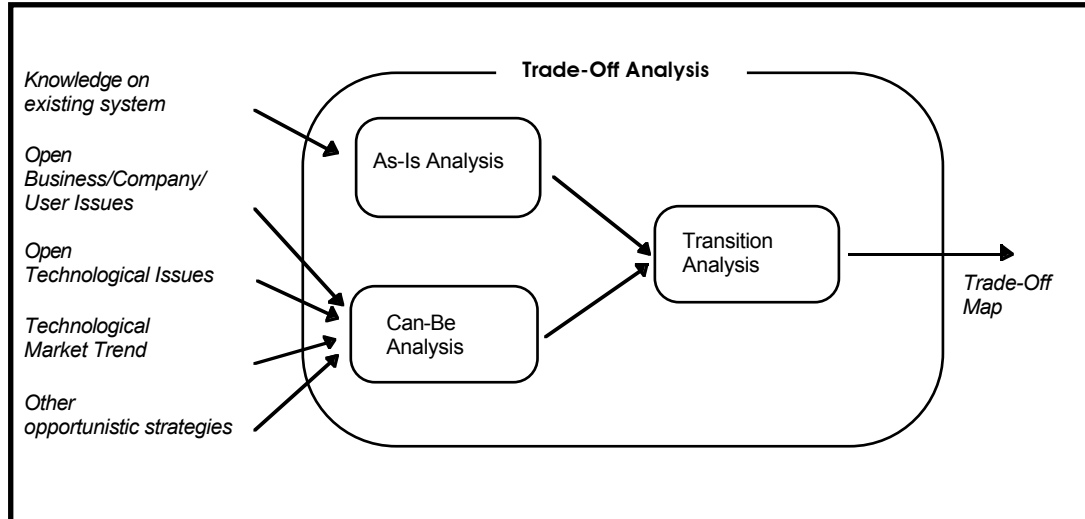


Figure G : Trade-Off Analysis Framework

The Trade-Off Analysis, defined in figure g, precedes every activities. Its aim is to address open technological issues and to examine the current technological market trend, helping in understanding and even anticipating the complex interrelationships surrounding changes.

Both the existing system and the desired one are assessed only on the basis of provided and desired technologies and open issues, in order to analyse possible transitions.

In other words, the Trade-Off Analysis focuses on the business benefits (external characteristics) rather than internal characteristics of a software system, and it involves analysis of both the existing software system and the possible future one as a black box.

On the basis of the result of this phase, it will be easier to understand and suggest which technologies should be replaced, and which technologies should replace the legacy ones; in other words, it will be stated whether a technology trade-off is suggested or not.

Role categories involved are: Strategic, Operational, Service.

2.4.1.3 Issue Assessment

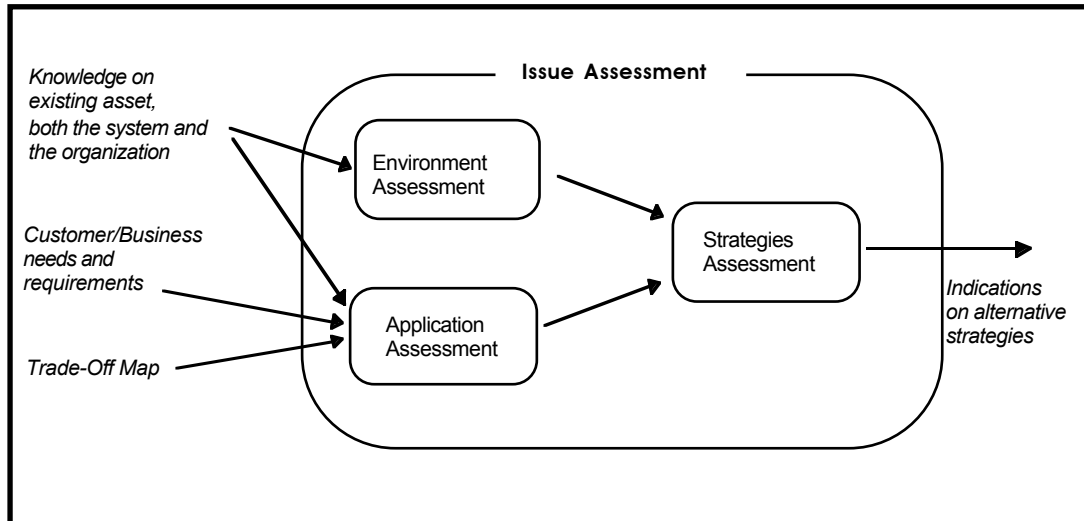


Figure H : Issue Assessment Framework

The Issue Assessment phase, detailed in figure h, involves internal understanding of the software system, that is white box analysis.

The initial activity of this phase is to establish both scope and direction of the reengineering effort. Inputs to this phases are:

- outputs of Trade-Off Analysis phase;
- knowledge of the legacy asset (both of the system and the organization);
- customer needs and requirements, and general business goals.

The output is the result of the assessment of the legacy asset with indications on candidate (or possible applicable) reengineering strategies to adopt. The different indications on the nature of the intervention to be performed are passed to the following phases. As shown in the figure above, Issue Assessment includes the following main activities:

- *Environment Assessment* is the phase of evaluation of the application environment. The aim of this phase is to assess the organization's level of preparedness to reengineer. The current environment and practice is evaluated in order to decide if the company is able to drive a reengineering activity; the maturity of the company's process for the application domain is evaluated as well.
- *Application Assessment* is the phase of evaluation of the needs of the application.
- *Strategies Assessment* is the phase of formulating possible alternatives for the work to be done on the current application.

Role categories involved are: Operational, Service.

2.4.1.4 Decision Analysis

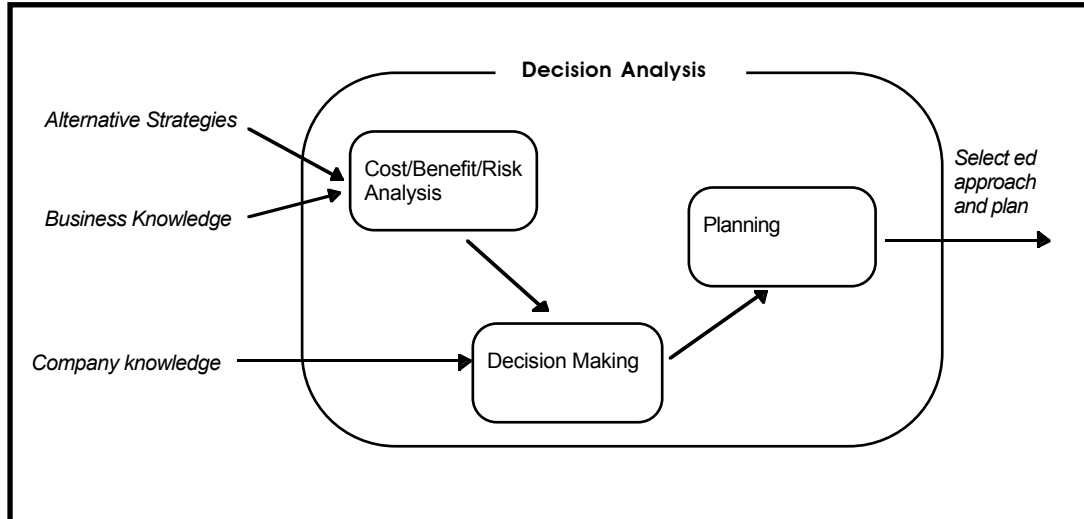


Figure I : Decision Analysis Framework

The Decision Analysis phase follows the Issue Assessment, and it is detailed in figure i. A robust decision-making process is needed to account for all the issues that can influence and affect reengineering. As integral part of the decision process, a cost/benefit/risk analysis must be performed, and a decision on the nature of the intervention must be taken. The output of this phase is a decision on the reengineering approach to be used, and a plan for the intervention. The best approach is selected among the Alternative Strategies in input.

In other words, the combined effect of Trade-Off Analysis and Issue Assessment (which could be grouped within a common activity, like *Investigate*) and Decision Analysis is to provide a systematic means for

- obtaining a thorough understanding of all the issues in reengineering the system;
- making an informed decision as to whether or not to proceed with reengineering;
- developing a project plan to drive the project of implementing the chosen reengineering strategy.

Role categories involved are: Strategic, Operational.

2.4.1.5 Solution Implementation

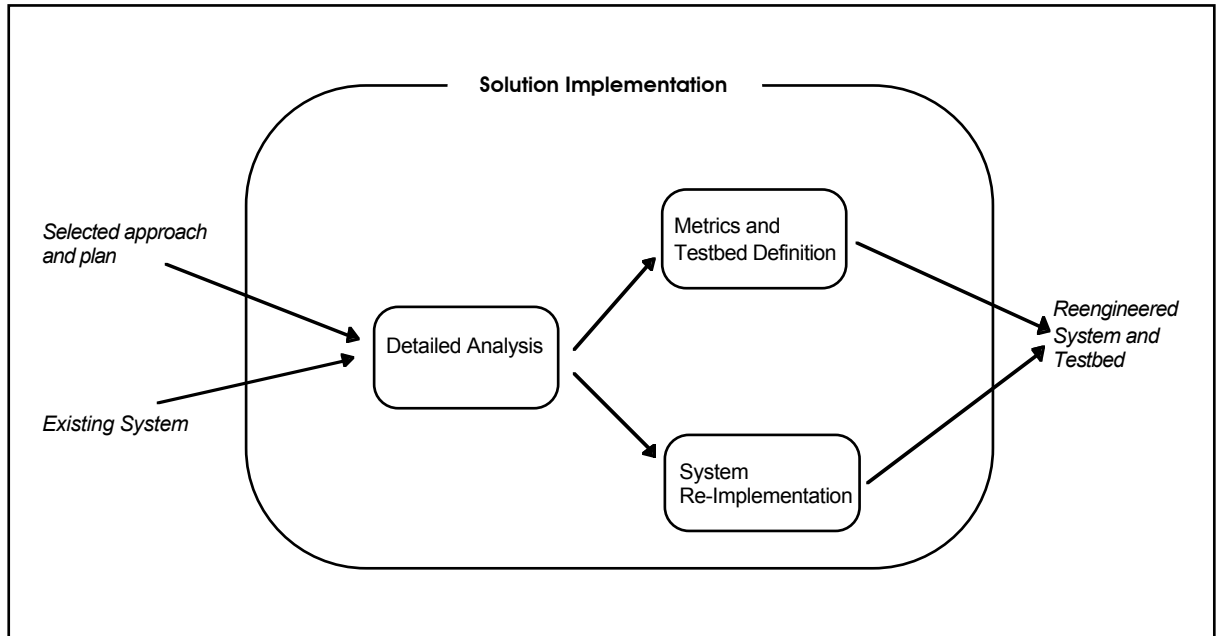


Figure J : Solution Implementation Framework

The Solution Implementation phase (figure j) constitutes the core element of the reengineering effort, and models the system transition phase.

This phase of the framework is expanded as follows:

- the initial activity consists in deeply understanding the existing application, including its components, structure and, if any, reusable components; that is to perform a detailed analysis of the existing system;
- a validation suite or testbed, and also metrics, must be prepared for assessing the new system;
- the final activity re-implements the system.

Role categories involved are: Operational, Service.

2.4.1.6 Solution Deployment

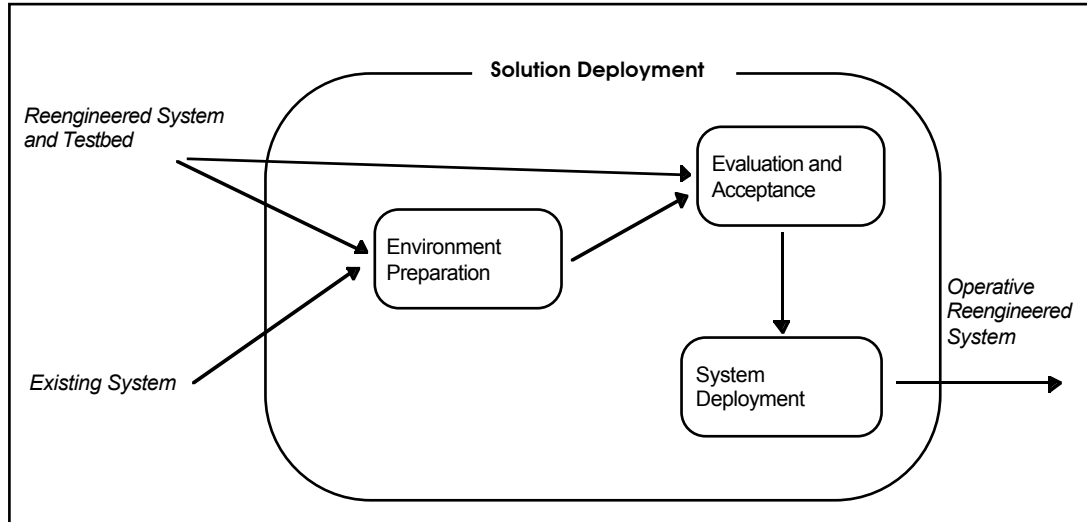


Figure K : Solution Deployment Framework

The Solution Deployment phase of the reengineering process framework models the deployment of the reengineered system to the new operational environment at the user site. It is detailed in figure k.

Once a solution has been validated and accepted, it must be deployed to its operational environment. This means that:

- the new delivered system is validated on a predefined testbed or validation suite in order to accept or refuse it (and metrics must be computed to evaluate the quality of the new system);
- support material must be prepared if users need to be trained on the new system, and the operational environment must be prepared for the deployment of the new system;
- the new system, if accepted, is deployed to the new operational environment.

Role categories involved are: Service.

2.4.1.7 Kaizen Improvement

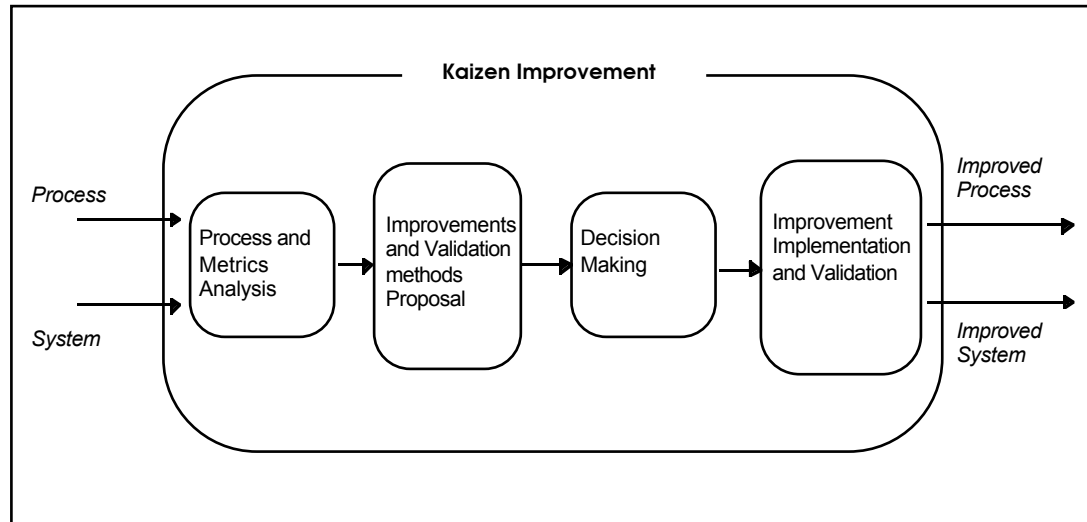


Figure L : Kaizen Improvement Framework

The final phase of the reengineering process framework is the continuous, or Kaizen, improvement of the process itself (figure 1). This puts a link from the end of the current project and the next project's beginning, and models the way in which reengineering is seen as an enabling technology for building evolutionary applications.

This means two things:

- the reengineered system must be continuously refined to answer to new requirements;
- the reengineering process itself must be continuously refined.

From this point of view, this phase must not be just the last phase of the overall process: improvement must be seen as a pervasive activity, which continuously rethink of the current activity with the aim of improving it and the overall process. This is the reason it appears in the middle of the global framework picture, with links to every phases of the process.

As shown in the figure, the Kaizen Improvement can be modelled with the following sequence of intuitive activities:

- Process and Metrics Analysis
- Formulation of Methods for Improvements and their Evaluation
- Improvements Evaluation
- Improvements Deployment and Validation

Role categories involved are: Strategic, Operational, Service.

2.4.2 Summary of RENAISSANCE objectives

As discussed in previous sections, we have to take into account two main activities, named *What To Do* and *How To Do*, and three views of the subject legacy system: *Technical*, *Economic*, and *Managerial*.

table d summarizes which views will be supported by the method.

Details can be found in the previous discussion.

	What To Do	How To Do
<i>Technical View</i>	Supported	Supported
<i>Economic View</i>	Supported	Out of scope
<i>Managerial View</i>	Supported	Out of scope

Table D : Summary of RENAISSANCE objectives

The table can be informally read as follows.

On the basis of this framework, RENAISSANCE defines a method for evaluating the reengineering needs of a legacy software system through a combination of technical, economic and managerial assessments, and for implementing the selected reengineering strategy. The result is a reengineered software system, which is able to evolve, from the technical point of view, both in the short and long term.

Given that the *How To Do* aspects of the economic and managerial views are widely covered in the BPR literature, the method developed by the RENAISSANCE project will not cover these aspects, and we refer the reader to the excellent material available elsewhere (i.e. [HAMM 93], [DAVI 93], [MANGA 94], [BERG 96]).

3 Bibliography

- [ALAN 95] Alan E. Giles and Dennis Barney, "Metrics Tools: Software Cost Estimation", STSC Crosstalk, June 1995.
- [AMES 94] AMES Consortium, AMES ESPRIT project #8156, "AMES Methodology and Process Model", 1994
- [ARN 93] Arnold, R. S., "Software Reengineering", IEEE Computer Society Press, CA, 1993
- [BENN 95] Bennet, K., "Legacy Systems: coping with success", IEEE Software, January 1995
- [BERG 96] Bergey, J. K., S. R. Tilley, et al. (1996). An Enterprise Perspective of Reengineering, Software Engineering Institute, Carnegie Mellon University.
- [BOEH 81] Boehm Barry, "Software Engineering Economics", Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- [BUSH 90] Bush, E., "A CASE for existing system", Language Technology White Paper, Salem, MA, 1990
- [CAPE 96] Capers Jones, "Activity-based software costing", Software Challenges Computer, Vol. 29, No. 5, May 1996.
- [COU 94] Coulouris, George, et. al., "Distributed Systems - Concepts and Design 2nd ed.", Addison-Wesley, 1994.
- [CSC 96] Computer Science Corporation, "Legacy Asset Management: Setting a Course from the Past to the Future", Technical Report, May 1996
- [DAVI 93] Davidson W. H., "Beyond Re-Engineering: The Three Phases of Business Transformation", IBM Systems Journal, Vol. 32, No. 1, January 1993

- [DOD 97] "Software Reengineering Handbook: Version 3", US Department of Defense, March 1997
- [FENT 91] Fenton, N. E. (1991). Software Metrics, A Rigorous Approach, Chapman & Hall.
- [GAL 95] Gall, Harald C., et. al., "Architectural Transformation of Legacy Systems", 17th International Conference on Software Engineering, Seattle, Washington, U.S.A, April 1995.
- [GANTI 95] Ganti, N., Brayman, W., "The transition of Legacy Systems to a Distributed Architecture", John Wiley & Sons, 1995
- [GARM 96] Garmus, D. and D. Herron (1996). Measuring the Software Process - A Practical Guide to Function Points, Yourdon Press.
- [GRE 85] Green, M., "Report on Dialogue-Specification Tools", Springer-Verlag New York, 1985, pp. 9-20.
- [HAMM 86] Hammer, M., "Reengineering Work: Don't Automate, Obliterate", Harvard Business Review, 1990
- [HAMM 93] Hammer, M., Champy J., "Reengineering the Corporation", Harper Collins, New York, 1993
- [IMAI 86] Imai, M., "Kaizen: The Key to Japan's Competitive Success", McGraw-Hill Publishing Company, New York, 1986
- [KAR 95] Karlsson, Even André, et. al., "Software Reuse – A Holistic Approach", Wiley, 1995.
- [LEHM 80] Lehman, M. M., "Programs, Life-Cycles, and the Laws of Program Evolution", Proc. IEEE, 1980
- [LOND 87] Londeix, Bernard, "Cost Estimation for Software Development", Addison Wesley, 1987.
- [MANGA 93] Manganelli R. L., Klein M.M., "The Reengineering Handbook: A Step-by-Step Guide to Business Transformation", AMACON, 1994
- [MCCL 90] McClure, C. L., "The Three Rs of Software Automation: Re-engineering, Repositories, Reusability", Extended Intelligence, Inc., Chicago, IL, 1990
- [MIL 95] MIL-HDBK-171, Work Breakdown Structure for Software Element, July 1995.
- [MOAD 90] Moad, J., "Maintaining the Competitive Edge", Datamation, February 1990
- [MUL 95] Müller, Hausi A., Tutorial "Understanding Software Systems using Reverse Engineering Technologies", 17th International Conference on Software Engineering, Seattle, Washington, U.S.A, April 1995.
- [REL 90] Reliability, C. f. S. (1990). Software Reliability Handbook, Elsevier.
- [ROCH 91] Rochester, J. B. and D. P. Douglass. (1991.). "Reengineering Existing Systems." I/S Analyzer, Vol. 29,(No. 10.): pp. 1-12.

-
- [SEI 96] SEI (1996). Assessing the Evolvability of a Legacy System, Software Engineering Institute, Carnegie Mellon University.
- [SNEE 94] Sneed, H. and E. Nyary (1994). "Downsizing Large Application Programs." *Software Maintenance: Research and Practice* 6(6): 235-247.
- [SNEE 95] Sneed, H. M. (1995 January). "Planning the Reengineering of Legacy Systems." *IEEE Software* 12(1): pp.24-34.
- [SNEE 91] Sneed, H. M. (September 1991). "Economics of Software Reengineering." *Journal of Software Maintenance: Research Practice* Vol. 3: pp. 163-182
- [SRHA 95] American DoD Technical Report, JLC-HDBK-SRAH, "Software Reengineering Assessment Handbook", Volume I, II, Version 2.0, 1995
- [STO 97] Storey, Margaret-Anne D., et. al., "Manipulating and Documenting Software Structures", World Scientific Publishing Co., in press
- [STSC 93] Software Technology Support Center, "Reengineering Technology Report", STSC Hill AFB, UT, 1993
- [TILL 95] Tilley, S., "Perspectives on Legacy Systems Reengineering", Software Engineering Institute, Reengineering Center, Draft Version 0.3, 1995
- [TRY 97] Tryggeseth, Eirik, "Support for Understanding in Software Maintenance", PhD Thesis, NTNU, Trondheim, Norway, March 1997.
- [WON 96] Wong, K., "Rigi Blurb", <http://www.rigi.csc.uvic.ca/>, February 1996.
- [WOOD 92] Wood Michael, "A Reengineering Economics Model" STSC Crosstalk, June 1992.
- [YOU 89] Yourdon, Ed, "RE-3 - Part 1", *American Programmer*, Vol. 2, No. 4, April 1989, pp. 3-10.