Reengineering Study
Chapter 5
5.0 Reengineering

The term Re-engineering is quickly becoming a favored buzz word, but what does it mean to software managers and developers? Basically, re-engineering is taking existing legacy software that has become expensive to maintain or whose system architecture or implementation are obsolete, and redoing it with current software and/or hardware technology or technically it is the examination, analysis, and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The difficulty lies in the understanding of the existing system. Usually requirements, design and code documentation is no longer available, or is very out of date, so it is unclear what functions are to be moved.

Often the system contains functions that are no longer needed, and those should not be moved to the new system. This thesis first defines re-engineering and looks at why it is necessary, and what are the objectives. The steps, phases and approaches are then discussed. A unique method of reengineering, Hybrid re-engineering, is defined and discussed, followed by experience reports from industry: what potentially worked and what didn't.

5.1 What is Re-engineering?

Re-engineering is the examination, analysis and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The process typically encompasses a combination of other processes such as:

5.1.1 Reverse Engineering,

5.1.2 Redocumentation,

5.1.3 Restructuring,
5.1.4 Translation, and
5.1.5 Forward Engineering.

The goal is to understand the existing software (specification, design, implementation) and then to re-implement it to improve the system's functionality, performance or implementation. The objective is to maintain the existing functionality and prepare for functionality to be added later.

5.2 Re-engineering Objectives

Although objectives of a specific software re-engineering task are determined by the goals of the owners and users of a system, there are two general re-engineering objectives

5.2.1 Improve quality — Typically, the existing software system is of low quality, due to many modifications. User and system documentation is often out-of-date or no longer in existence. Re-engineering is intended to improve software quality and to produce current documentation. Improved quality is needed to increase reliability, to improve maintainability, to reduce the cost of maintenance, and to prepare for functional enhancement. Object-oriented technology may be applied as a means for improving maintainability and reducing costs.

5.2.2 Migration — Old working software may still meet users’ needs, but it may be based on hardware platforms, operating systems, or languages that have become obsolete and thus may need to be re-engineered, transporting the software to a newer platform or language. Migration may involve extensive redesign if the new supporting platforms and operating systems are very different from the original, such as the move from a mainframe to a network-based computing environment.
However, the number of large systems being built from scratch is diminishing, while the number of legacy systems in use is very high. While the functionality of existing systems remains constant, the context of new systems, such as the application environment, system level hardware and software, are different. Enhancements to the functionality of the existing systems may also be needed, but although the re-engineering effort may configured for enhancements, they should not be incorporated until after the re-engineering is complete. This allows for comparison of functionality between the existing system and the new system.

The problem is that systems currently in use, "legacy" systems, have become lacking in good design structure and code organization, making changes to the software difficult and costly. Corporations do not want to "trash" these systems because there are many built in subtle business application processes that have evolved over time that would be lost. Often the developers of the legacy systems are not available to verify or explain this information; the only source is the current software code. The original expense of developing the logic and components of the software systems should not be wasted, so reuse through re-engineering is desired.

The challenge in software re-engineering is to take existing systems and instill good software development methods and properties, generating a new target system that maintains the required functionality while applying new technologies. Although specific objectives of a re-engineering task are determined by the goals of the corporations, there are four general re-engineering objectives:

5.2.3.1 Preparation For Functional Enhancement
The re-engineered target system can be built to easily facilitate the enhancements. For example, if the desired system enhancements build on object-oriented design, the target system can be developed using object-oriented technology in preparation for increasing the functionality of the legacy system.

5.2.3.2 Improve Maintainability
As systems grow and evolve, maintainability becomes a bottleneck and time consuming. An objective of re-engineering is to re-design the system with more appropriately functional modules and explicit interfaces. Documentation, internal and external, will also be current, hence improving maintainability.

5.2.3.3 Migration
The computer industry continues to grow at a fast rate, new hardware and software systems include new features, quickly outdating current systems. For these reasons, companies with working software that meets their needs might need to migrate to a newer hardware platform, operating system, or language.

5.2.3.4 Improve Reliability
Although it is possible that the reliability never was very high, more likely, over time and with multiple changes, there have been problems, one change causing multiple additional problems. As maintenance and changes continue, the reliability of the software steadily decreases to the point of unacceptable.
Although re-engineering should not be done to enhance the functionality of an existing system, it is often used in preparation for enhancement. Legacy systems, through years of modifications due to errors or enhancements, become difficult and expensive to change. The code no longer has a clear, logical structure and documentation may not exist, and if it exists, it is often outdated. Re-engineering specifies the characteristics of the existing system that can be compared to the specifications of the characteristics of the desired system. The re-engineered target system can be built to easily facilitate the enhancements. For example, if the desired system enhancements build on object-oriented design, the target system can be developed using object-oriented technology in preparation for increasing the functionality of the legacy system. As systems grow and evolve, maintainability costs increase because changes become difficult and time consuming. An objective of re-engineering is to re-design the system with more appropriately functional modules and explicit interfaces. Documentation, internal and external, will also be current, hence improving maintainability.

The computer industry continues to grow at a fast rate; new hardware and software systems include new features, quickly outdating current systems. As these systems change, personnel skills migrate to the newer technologies, leaving fewer people to maintain the older systems. In a relatively short time, manufacturers no longer support the software and hardware parts become expensive. Even more important is the compatibility of the older systems with the newer ones.

For these reasons, companies with working software that meets their needs might need to migrate to a newer hardware platform, operating system, or language. The fourth objective of re-engineering is to achieve greater reliability. Although it is possible that the reliability never was very high, more likely, over time and with
multiple changes, there have been "ripple effects", one change causing multiple additional problems. As maintenance and changes continue, the reliability of the software steadily decreases to the point of unacceptable.

5.3 Re-engineering Concepts

5.3.1 Software Development Levels of Abstraction

Levels of Abstraction, Figure 5.1, that underlie the software development process also underlie the reengineering process. Each level corresponds to a phase in the development life cycle and defines the software system at a particular level of detail (or abstraction).

![Figure 5.1. Levels of Abstraction]

The conceptual abstraction level is the highest level of abstraction. Here the concept of the system - its reason for existence - is described. At this level functional characteristics are described only in general terms. In the requirement abstraction level functional characteristics of a system are described in detailed terms. In these first two levels internal system details are not mentioned. In the design abstraction level system characteristics such as architectural structure, system components, interfaces between components, algorithmic procedures and data structures are described. The implementation abstraction level is the lowest
level. Here a system description focuses on implementation characteristics and is represented in a language understood by a computer.

5.3.2 General Model For Software Re-engineering

Re-engineering starts with the source code of an existing legacy system and concludes with the source code of a target system. This process may be as simple as using a code translation tool to translate the code from one language to another (FORTRAN to C) or from one operating system to another (UNIX to DOS). On the other hand, the re-engineering task may be very complex, using the existing source code to recreate the design, identify the requirements in the existing system then compare them to current requirements, removing those no longer applicable, restructure and redesign the system (using object-oriented design), and finally code the new target system. Figure 5.2 depicts a general model for software re-engineering that indicates the processes for all levels of re-engineering based on the levels of abstraction used in software development.

---

**Figure 5.2. General Model for Software Reengineering**

The model in Figure 5.2 applies three principles of re-engineering: abstraction, alteration, and refinement. Abstraction is a gradual increase in the abstract level of...
System representation is created by the successive replacement of existing detailed information with information that is more abstract. Abstraction produces a representation that emphasizes certain system characteristics by suppressing information about others. This upward movement is termed reverse engineering and has associated sub-processes, tools and techniques. Alteration is the making of one or more changes to a system representation without changing the degree of abstraction, including addition, deletion and modification of information, but not functionality. Refinement is the gradual decrease in the abstraction level of system representation and is caused by the successive replacement of existing system information with more detailed information. This is termed forward engineering and resembles software development of new code, but with some process refinements.

To alter a system characteristic, work is done at the level of abstraction at which information about that characteristic is explicitly expressed. To translate the existing code to a target language no reverse engineering is needed, the alteration (recoding) is done at the implementation level. As the level of abstraction increases, the alteration tasks change and the amount and tasks of reverse engineering change. To re-specify requirements, reverse engineering techniques must be applied to the implementation and to the design to obtain the functional characteristics.

5.4 Software Reliability Methods

5.4.1 Introduction

"Software reliability is the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence
of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered" (ANSI/IEEE Std 729-1983-1). From this definition it is observed that the failure behaviour of a software is a function that depends on:

5.4.1.1 Number of faults in the software; and
5.4.1.2 Execution profile

Thus, the reliability of a software is a function that combines number of faults and probability of these faults to occur, i.e. to produce a failure. In other words, the probability that the fault exists and the faulty code is executed. As a consequence, it can be improved by reducing either the number of faults or the probability of a fault to produce a failure (focusing our testing effort on the most frequently used functionality).

"Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment."

Techniques to improve the reliability of a software by reducing the number of faults are classified into three different groups:

5.4.1.3.1 Fault avoidance is aimed at preventing the introduction of faults during the development of the software. In this group are included all the techniques that look into the process of software development: standards, methodologies, etc. The techniques within this group are process oriented;

5.4.1.3.2 Fault detection is aimed at detecting faults once the code has been developed. These techniques focus on the product obtained rather than in the process. These techniques are product oriented; and
5.4.1.3.3 Fault tolerance is aimed at giving a controlled response for those uncovered faults. These techniques are used in safety-critical and high availability systems.

5.5 Nature of Faults

The aim here is to understand where in the process of software development (from the concept definition to the disposal of the system), the Faults are most likely to be introduced and how. Once the nature of the Faults is understood, it is possible to better control the process in such a way that the introduction of Faults can be minimized and the performance of the different techniques to find the Faults remaining in the product improved. In order to minimize the introduction of Faults in the system, it is important to understand where and how in the process are the Faults introduced with higher probability. Which parts of the process are more suitable for the Faults to be introduced and why.

Faults are likely to be introduced in every phase in the Project and they are propagated between phases. Whenever the natural language (or any other source of imprecision) is used, ambiguity can be introduced. Fault avoidance techniques focus on finding where the faults are introduced and are aimed at providing standards, methodologies or impositions that prevent the developer from introducing faults.

On the other hand, knowing where the faults are more suitable to be introduced, increase the probability of success of those techniques and tools that focus on finding defects. Faults can be introduced:
5.5.1 During the requirements specification. Good requirements are those that are necessary, verifiable and attainable. A poor requirements elicitation process will not detect many requirements that are not clearly stated and as a consequence will not be properly interpreted by designers, implementers or IVV teams, thus resulting in a Failure. Firesmith presents an extensive list of attributes that good quality requirements must exhibit, namely: cohesiveness, completeness, consistency, correctness, currency, customer/user orientation, external observability, feasibility, lack of ambiguity, mandatory, metadata, relevance, usability, validatability and verifiability. Hooks [75] gives some recommendations to specify requirements still using natural language (English) but with some kind of formalism. Hooks also presents the most common problems in writing requirements:

5.5.1.2 Making bad assumptions;
5.5.1.3 Writing implementation (how) instead of requirements (what);
5.5.1.4 Describing operations instead of writing requirements;
5.5.1.5 Using incorrect terms;
5.5.1.6 Using incorrect sentence structure or bad grammar;
5.5.1.7 Missing requirements; and
5.5.1.8 Over-specifying.

High-quality requirements is the first step to produce high-quality software. A good requirement should be clearly stated. It is important that the requirement is not misunderstood, it must be unambiguous. If requirements documents are inconsistent, ambiguous, incomplete or subject to misinterpretation, the
development (costs, schedules, ...) and the final product will be affected. Reviews are an effective way of finding problems but they result very expensive because they involve a number of people spending time reading and checking the requirements document. Two approaches to this problem can be followed:

5.5.1.9.1 Restrict the language used to specify requirements (restricted subset of English or even formal languages like Z); and

5.5.1.9.2 Analyse the requirements and identify potential problems in the specification.

The first approach has the advantage of a language without ambiguities but on the other hand, reduces the freedom to specify requirements. Moreover, the language used to specify must be known across all the involved parties (users, designers, IVV teams, ...) and this is normally not the case. The second approach has the advantage that the user has all the expressiveness from Natural Language and the document can be shared and reviewed by all the involved parties. Since reviews are very expensive and often of limited effectiveness, these reviews can be automated by implementing a tool to verify requirements. This tool should check three important aspects of quality attributes:

5.5.2 Expressiveness: characteristics dealing with the understanding of the meaning of the requirements (ambiguities, readability of the document, ...);

5.5.3 Completeness: characteristics dealing with the lack of information necessary in the specification; and
5.5.4 **Consistency:** characteristics dealing with semantic contradictions in the document.

There are some tools available but they only provide expressiveness analysis like ARM (Automated Analysis of Requirement Specifications). Some tools like QuARS provide additional support to consistency and completeness analysis but it still must be performed by the reviewer.

5.5.5 **During The Design Phase.** Lack of trade-offs, software budget reports or the use of new technologies are common sources of problems introduced within this phase. Important decisions are taken during the design phase and they should be well founded. It is important to demonstrate the feasibility of the proposed solution before going further in the implementation. Formal specification is a solid approach to validate designs;

5.5.6 **During The Coding Phase.** Many faults are introduced during this phase depending on the experience and workload of the staff, complexity of the algorithms, etc. A first check in this phase is performed by compilers which check for lexical and syntactical consistency. Static analysers of dynamic properties have been implemented but with still important limitations. The idea behind this concept is the static validation of properties exhibited during the runtime like access to areas of memory de-referenced, variables not initialized, etc; and

5.5.7 **During The Maintenance Phase.** As the software is modified during the maintenance (due to corrective maintenance or due to the implementation of new features), faults can be introduced by re-
engineering (new requirements, design, coding, ...). Another important source of faults in this stage, is a poor configuration control and poor integration testing (regression testing is very important to verify that new bugs have not been introduced). The reliability in this stage varies as faults are removed and new bugs are introduced.

According to the way they manifest, the faults can be classified into three categories:

5.5.8.1 **Transient Faults** are those that appear for a short period of time and then disappear after a while. They are mainly produced by the failure of processors or by transient or permanent interference in the communication subsystem (network faults). Against these failures some of the techniques described further on in this book can be implemented;

5.5.8.2 **Permanent Faults** are those that will remain in the system until the fault has been removed, for example software design defects; and

5.5.8.3 **Intermittent Faults** appear and disappear from time to time.

ODC (Orthogonal Defect Classification) is a defects classification schema that helps to better understand the problems introduced in every phase of the development. The following attributes are collected either when the defect is identified or when it is fixed: activity, trigger, impact, type, qualifier, source and age. With these attributes it is possible to produce quality metrics that can be used to measure the customer satisfaction.
Software reliability must be integrated in the whole life cycle and must involve all the staff participating in the project. It cannot be just considered at the end. State of the art techniques to improve the software reliability are described in this section. All these techniques are classified into three different groups: Fault avoidance, Fault detection and Fault tolerance. As introduced before, in order to increase the reliability of the software it is possible to reduce the number of faults and/or reduce the probability of a fault to cause a failure (i.e., to be executed). With the classification here described and aimed at reducing the failure behaviour of the target system, the process can be optimised by:

- Avoiding the introduction of faults (standards, methodologies, ...);
- Implementing in our process techniques aimed at finding and eliminating the faults (since it is not possible to produce fault free software);
- Implementing fault tolerance if the nature of the software requires it (since it is not possible to guarantee that there are no faults remaining in our software); and
- Using operational profiles as representative as possible to optimise the effort and the effectiveness of our testing process.

5.6 Fault Avoidance

The techniques in this category focus on the process. They are aimed at defining methodologies and standards that control the development process and prevent the introduction of faults. All the known methodologies and standards like ISO9000, CMM, ECSS, DO178B, etc. are included here. In principle it could be stated that the higher the CMM level the company has, the lower the probability to introduce
faults in the software. This refers mainly to those problems introduced for example by poor configuration control, poor requirements elicitation process, etc. The maturity level of the company is an indicator of the control it has over the software development process. There are studies that try to assess the reliability of the final product (a first approximation) based on the development process aided by the use of Bayesian networks. They establish a framework to assess this reliability based on the confidence in the development process. This approach is very subjective but could give a rough indication before the product has been developed.

Here we review several techniques that can be introduced in the development process to prevent the introduction of faults:

5.6.1 **Formal Methods** are design techniques that use rigorous mathematical models to build software and hardware systems and help to reduce the errors introduced into a system, especially at the earlier stages of design. Formal specifications use mathematical notation to describe in a precise way the properties that a software must have, without constraining the way this is achieved. The notation to be used should allow the representation of both static and dynamic properties of the system to be built.

Static aspects are:

5.6.2.1 The states the system can occupy; and
5.6.2.2 Invariant relationships when moving from one state to another state.

And the dynamic aspects are:

5.6.2.3 The operations that are allowed;
5.6.2.4 The relationship between inputs and outputs; and
Formal methods are used during the specification, design and construction of software products. The major problem that Formal methods encounter is that they offer a very limited expressiveness compared to the Natural language and they are still rather difficult to apply. They provide a precise way to represent the system but require a big effort to use and understand the notation.

5.6.2.6 **Software Hardening.** It has been demonstrated that the complexity of a software has a direct repercussion in the number of faults introduced. Complex software is difficult to understand and to maintain. The more complex a software is, the more the probability of faults to be introduced and to be uncovered during testing. Based on this principle, Software hardening tries to limit the complexity of the software by establishing reference values for a set of measurements that provide a quantification of software complexity. Every system in the nature describes a process that can be characterized by a set of measurements quantifying the complexity of the system.

These measurements can be equally applied to characterize the complexity of a software system. After that, limits for these parameters are set in such a way that the complexity of a software does not exceed our ability to handle it. It has been conlaimated the limitation to handle software over certain limits of complexity. These limits differ depending on the maturity and specific knowledge of the staff. The following measurements of this complexity that will be later on extrapolated to software systems are:
5.6.2.6.1 Volume;
5.6.2.6.2 Entropy; and
5.6.2.6.3 Temperature.

5.6.3 **Software Aging And Rejuvenation.** Software also suffers from degradation during its life. This happens because some bugs have not been detected and introduce memory leaks or other problems that are accumulated and will probably produce a failure after a certain execution period. This is known as software aging and is no other than a degradation of the execution of a software mainly by memory corruption.

It has been conclamated that many of these failures can be prevented just by restarting the application periodically. This introduces the concept of software rejuvenation. This degradation of the execution grows with the execution time and affects also the reliability of a software. As mentioned before, the reliability of a software has a time component. Even if the fault is still there, the failure intensity can be reduced just by restarting the system and reducing the probability of it becoming a failure. The overall reliability of a system and therefore the customer satisfaction will increase.

The system can be totally or partly restarted and this restart can be scheduled according to the specific behaviour of the software and the availability requirements of the different components. The
scheduling activity is the heart of this technique and has to be good enough to restart the component before it fails and within the availability constraints of the system (especially difficult in systems with 24x7 availability).

5.6.4 Use And Certification Of COTS. The use of Commercial Off The Shelf (COTS) applications is very attractive since they can save costs in both development and testing. But an important aspect of this re-use in the context of software reliability is the certification of these COTS for the new software. It is very likely that those COTS that are intended for re-use have been developed to meet different requirements and thus will not meet our specific reliability requirements. Many techniques have been described to improve their reliability by introducing new layers between these COTS and the applications.

These new layers observe the data flowing through the interfaces and modify or reject data likely to cause harm. In many cases, an important effort is necessary to accommodate these products in the new software reducing the benefits of this software re-use. The technique of Generic Software Wrappers can provide a common framework for integrating COTS without modifying the software and in such a way that they are protected (they run in kernel space) and non-bypassable.
5.7 Fault Detection

The techniques in this category are aimed at detecting faults once the code has been developed. These techniques focus on the product obtained rather than in the process.

5.8.1 Formal Methods and Formal Verification. If formal methods have been used to specify the system, it is possible to verify different properties that the software should exhibit or not exhibit. This is an important attribute that can be verified with formal methods;

5.8.2 Dynamic Analysis is performed during the execution of a program. This technique usually instrument the code, get some statistics and look into the dynamic behaviour of the software at runtime. The problem associated to this analysis is that it is based on the execution paths selected in the test cases;

5.8.3 Semantic Analysis performs a static analysis of dynamic properties of the software. The idea behind this concept is the static validation of properties exhibited during the run-time like access to areas of memory de-referenced, variables not initialized, etc. PolySpace Technologies has developed a tool detecting run-time errors implementing Semantic Analysis. The following run-time errors can be detected through the use of Semantic Analysis:

5.8.4 Concurrent accesses to shared data;
5.8.5 Pointer de-referencing issues (null, out-of bounds accesses);
5.8.6 Out-of-bounds array accesses;
5.8.7 Read accesses to non-initialized data;
5.8.8 Invalid arithmetic operations (division by zero, square root of negative numbers, etc.);
5.8.9 Float, integer overflow/underflow;
5.8.10 Illegal type conversion (for example float $\rightarrow$ int, long $\rightarrow$ short);
5.8.11 Dynamically unreachable code;
5.8.12 Non-termination of loops; and
5.8.13 Initialized return values.

Semantic Analysis is a mathematical approach that statically analyze the dynamic properties of software applications at compilation time (no need to execute the code).

5.8 Fault Tolerance
Fault tolerance is aimed at providing a controlled response, sometimes at a degraded level, upon uncovered faults in the system. Several levels of Fault tolerance can be implemented, namely:

5.9.1 Full Fault Tolerance, in which the system continues the normal operation when a failure occurs;
5.9.2 Graceful Degradation, in which the system operates with reduced functionality or performance; and
5.9.3 Safe Mode, where the system operates with a minimum functionality and performance during the time that the system is repaired and can be again fully operational.

Software Fault Tolerance techniques are divided in two groups: single version and multi-version software techniques. The first focus on improving the fault tolerance of a single software by adding mechanisms to detect, contain and handle errors caused by the activation of faults whilst multi-version software techniques use
multiple versions of a software in such a way that different versions of this software do not cause a failure at the same time, i.e. with different failure profiles.

Software Fault tolerance can be applied at any level: software component, full application, whole system including the operating system, etc. Single version fault tolerance has the advantage of developing only one version of a software but with added complexity and an important performance overhead. The disadvantage is that it relies in only one version to determine the output of the system. The critical issue is detecting internal errors based on the different checks implemented.

With multiple-version software the probability of a correct output increases (it remains controversial to what extent can a multiple-version software increase the reliability with respect to single version) but it is still difficult to implement the decision module which provides the correct output based on the outputs from the different versions (especially difficult when different outputs are valid). Another critical issue in multiple-version software fault tolerance is guaranteeing the independence of failure on multiple versions.

5.9 Input Sequence Workarounds

Torres-Pomales [70] defines a fourth technique to deal with software reliability. This is nothing more than accepting that the software has a particular design fault and considering this fault as a "feature" of the software. The operator uses an alternative input sequence that produces the expected results without activating the failure. This way the user works around the fault and the system is still available.
5.10 Development of Operational Profiles

It is really important that the testing efforts are concentrated in those functionalities with higher probability of execution. Operational profiles identify every operation in the system and the probability of these functions. These probabilities are then combined with the criticality of every operation to obtain a distribution of the testing effort between operations. The aim of development of operational profiles is an effective use of testing resources to effectively increase the software reliability.

5.11 Conclusions

Software reliability is in general very immature. Most of the techniques here reviewed are still difficult to apply or with a limited applicability. Software complexity is growing very fast and there is an urgent need for tools supporting the validation of systems with high reliability requirements. Software reliability is still to be integrated in the normal development cycle as any other integral process and cannot be just considered at the end.

Formal methods are promising in the avoidance of faults and provide support to verification but are still difficult to apply and suffer from state space explosion which limits its applicability. They are suitable for critical parts of a software and embedded applications of limited size. Natural language or semi-formal specifications are still preferred because they can be easily shared among the different parties involved in the development: users, designers, developers, customers, ...

The development of operational profiles effectively helps to:
5.11.1 Focus resources on the most used and/or most critical functions;
5.11.2 Maximize the test effectiveness by making the tests highly representative of the operational field

The benefits obtained with the development of operational profiles worth are the effort. An important issue in software reliability is the lack of tools supporting reviews:

5.11.3 Requirements Verification tools can highly improve the quality of requirements document without limiting the freedom when specifying requirements. The available implementations are very immature but still provide a great support to the user;

5.11.4 Static analysis of dynamic properties is very promising in the detection of runtime errors. However, this technique is still of limited applicability and demand lots of resources. Current implementations report very good results at commercial level.

In general there is an important niche for the development of tools capable to support analysis of commercial software and help the developers in the expensive and tedious process of requirements specification reviews. The implementation of Automated Requirements Verification Tools capable of checking consistency and completeness of requirements seems to be an important line of research. Such a tool could improve the quality of the specifications and at the same time reduce the costs due to reviews and defects escaping the reviews and detected at later stages with increased impact.

With systems getting more and more complex, especially safety critical and high availability systems, the fault tolerance should evolve in order to be able to solve
the problem of design diversity. Two issues remain critical in the area of Fault Tolerance Systems:

5.11.5 Design of output selection algorithm and
5.11.6 Failure independence in multiple-version software.

Important areas of research within Fault Tolerant Systems are:

5.11.7 Evaluation of independence between designs,
5.11.8 Semi-automatic generation of Fault Tolerant Systems and
5.11.9 Validation of Fault Tolerant Systems (path coverage).

A lot of work is still to be done to guarantee versions independence. Design diversity effectiveness still remains a controversial issue. Software reliability can be highly improved with the available techniques but as the software complexity grows, there is an increasing need of tools supporting the development of safety critical and high availability systems.

Several techniques have been presented in this section, that allow to improve the software reliability by:

5.11.10 **Defining Restrictions** in the architecture to limit the complexity and thus the probability to introduce faults;
5.11.11 **Optimizing The Test Efforts** to detect the most critical and probable failures (and thus increasing the customer satisfaction);
and
5.11.12 **Implementing Automated Verification** tools to highly improve the efficiency of reviews and testing activities
All these techniques are very valid and without doubts they considerably improve the reliability, though there is still a compromise between effectiveness and ease to apply. A selection or a combination of them is dependent on the reliability targets, experience of the staff involved in the project and nature of the project. Bearing in mind that software reliability has to be considered during the whole life cycle and integrated in the development as any other integral process (quality assurance, project management, ...), the selection must include a combination of fault avoidance techniques, fault detection techniques and if required, fault tolerance.

5.12 Re-engineering Phases and Tasks

There is a core process that every organization should follow when re-engineering. Reengineering poses its own technical challenges and without a comprehensive development process will waste time and money. Automation and tools can only support this process, not preempt it. The re-engineering process can be broken into five phases and associated tasks, starting with the initial phase of determining the feasibility and cost effectiveness of reengineering, and concluding with the transition to the new target system. These five reengineering development phases are:

5.12.1 Re-engineering team formation;
5.12.2 Project feasibility analysis;
5.12.3 Analysis and planning;
5.12.4 Re-engineering implementation; and
5.12.5 Transition and testing.

5.13 Re-engineering Team Formation
This team will manage the re-engineering effort from start to conclusion and will need comprehensive training in how to manage the technological change, the basics of re-engineering, and the use of target development and maintenance processes. Their tasks will be diverse, starting with establishing goals, strategies and an action plan within the current environment and based on the identified business needs including cost justifications. Although team members must have the "standard" software development skills, they will need additional, specific skills. They will be responsible for identifying, testing and purchasing new tools, then making sure personnel are properly trained on the tools and the tools are being effectively used. The team will need to provide internal marketing of the re-engineering work, consulting with personnel to verify the process is correctly being applied. These tasks will require team members to have good interpersonal skills to resolve rejection of new concepts and perceptions of software ownership. Since the field of re-engineering continues to evolve, the team members will also need to continue research in this technology.

5.14 Project Feasibility Analysis

The initial task of the re-engineering team is to evaluate the organizational needs and goals that the existing system meets. It is important that the re-engineering strategy fit with the organization's cultural norms. Software products currently in use must be analyzed in terms of problem specification including objectives, motivation, constraints and business rules. The value of the applications must be investigated to determine what is the expected return on investment from the re-engineering effort: the degree the software quality is expected to increase, the maintenance process efficiency improve, and the business value enhanced. Once the expectations are established, they must be expressed in a measurable way - reduction in cost of sustaining engineering, reduction in operations, improvement
in performance, etc. Then the costs of reengineering must be compared to the expected cost savings and the increase in value of the system.

5.15 Analysis and Planning

This re-engineering phase has three steps: analyze the legacy system, specify the characteristics of the target system, and create a standard testbed or validation suite to validate the correct transfer of functionality. The analysis step begins by locating all available code and documentation, including user manuals, design documents and requirement specifications. Once all of the information on the legacy system is collected, it is analyzed to identify its unique aspects. The current condition of the existing system, its maintainability and operability, must be specified in order to show a return on investment with the target system. A set of software metrics should be selected to assist in identifying the quality problems with the current system and the prioritization of applications that are candidates for re-engineering according to their technical quality and business value. The metrics should include measurements of the costs of changes to the software, if an increase in maintainability is one of the goals of the reengineering process. At the conclusion of the re-engineering effort the same metrics should be used to identify the quality of the new system and the return on investment. The collection of metrics on the new system should continue throughout development to identify if what is happening is normal or expected and in order to react quickly to abnormal signs.

Once the legacy system and its quality characteristics have been specified, the step of stating the desired characteristics and specification of the target system begins. The characteristics of the existing system that must be changed are specified, such as its operating system, hardware platform, design structure, and language. Finally,
A standard testbed and validation suite must be created. These will be used to prove the new system is functionally equivalent to the legacy system and to demonstrate functionality has remained unchanged after re-engineering. The testbed for the target system may be implemented incrementally if necessary, but traceability to the functions of the legacy system is important.

5.16 Re-engineering Implementation

Now that the re-engineering objectives have been specified, the approach has been defined, and the legacy system analyzed, the reverse and forward engineering are started. Using the Levels of Abstraction, the actual functions of the legacy system are unraveled by reverse engineering. Various tools are available for this task. These tools must be examined for usability in the context of the objectives of the reengineering process. They must integrate easily into the process with minimum massaging.

After the desired level of abstraction is reached, forward engineering can begin. Forward engineering corresponds exactly to the normal software development process, starting from the abstraction level reached in the reverse engineering process. That is, if the software is to be redesigned to fit a new overall system architecture, the reverse engineering process should extract the software requirements, and the forward engineering process would start with the development of a new design.

In forward engineering, any change or increase in functionality must be avoided since this complicates the validation process. Throughout this phase, quality assurance and configuration management disciplines and techniques must be
applied. Measurement techniques use should continue to assess the improvement to the software and to identify potential areas of risk.

5.17 Testing and Transition

As the functionality of the new system grows, testing must be done to detect errors introduced during re-engineering. The testing techniques and methods are the same as those used during a "from scratch" system development. Assuming the requirements for the new system are the same as those for the legacy system the test suite and test bed developed in the planning phase can be used. The same test cases can be applied to both the legacy system and the target system, comparing the results to validate the functionality of the target system. Software documentation on the legacy must be updated, rewritten or replaced during this phase so that they apply to the new system, and contain the information needed to operate and maintain it.

5.18 Hybrid Re-engineering

The SATC has coined the phrase "Hybrid Re-engineering" to mean a reengineering process that uses not just a single, but a combination of abstraction levels and alteration methods to transition an existing system to a target system. Projects doing hybrid re-engineering choose a combination of Abstraction levels based on the condition of the legacy system, the needs of the project and its budget and schedule.

In Hybrid Re-engineering, legacy systems are re-engineered using the approach shown in Figure 5.3, an adaptation of the General Model for Software Re-engineering. In Figure 5.3, three development tracks are utilized. The first track is a translation from existing code to a new language, operating system or hardware
platform with no abstraction. The second track uses the existing code to identify requirements that can be satisfied by the application of COTS packages. The third track is the more standard re-engineering process, the development of new code for project requirements that cannot be satisfied by either of the other tracks, and to "glue" together the translated and COTS components.

![Diagram of Hybrid Reengineering Tracks]

Re-engineering as a development methodology has inherent risks, such as schedule, functionality, cost, and quality. Hybrid re-engineering was developed to decrease some of these risks since COTS packages are expected to have high reliability and require minimal development time. Another method of decreasing time and cost through Hybrid re-engineering while maintaining functionality is through a straight translation of part of the current code to the new language or operating system.

Hybrid re-engineering is innovative, combining three distinct re-engineering efforts, hence the risks generally associated with re-engineering can increase by combining the risks inherent to each track. Since hybrid re-engineering is combining products from different development tracks (COTS, custom software and translated software), one new risk is the interface and interoperability of the products. For example, data transfer between products can cause compatibility and
timing problems; COTS packages may not work exactly as anticipated. In general, metrics can be used by management to improve software development efforts and minimize the risks. Metrics can indicate how well a project is meeting its goals. In hybrid reengineering, metrics can support the justification for decisions on track selection for different software functions and components.

5.18.1 Hybrid Re-engineering Tracks

The following sections describe each of the three hybrid re-engineering tracks. After each track is described, the risks associated with the track are identified and appropriate metrics defined.

5.18.1.1 Translation Track Hybrid Re-engineering

Figure 5.4 is a diagram of a "typical" software system that has been in use for some period of time. In this re-engineering example, we assume the project is moving from FORTRAN to C++ (but not necessarily to an object-oriented design).

![Figure 5.4. Current Software System](image)

In looking at the current software system, the manager sees two classifications of code, stable code that has had minimal modifications and whose requirements have remained unchanged, and some code that has undergone multiple changes and has become unstable, unreliable, and costly to maintain. Re-engineering the stable code may not require total reverse engineering; it might be feasible to simply re-code this portion into the new language or new environment. This process constitutes the translation track of hybrid re-engineering. In this track the code in the existing
system that is relatively stable, having had minimal changes to the original design and architecture, must be identified. This can be accomplished by an analysis of the code and of change reports. Prior to re-engineering, in identifying candidates for translation, metrics such as the logical and calling complexities provide valuable information on structure and coupling and suggest candidates for translation. In identifying components that have been extensively maintained, change or problem reports supply the data. Tracking the criticality of functions will assist in making tradeoff decisions.

**Figure 5.5. Transaction Track Hybrid Reengineering**

In the Translation track, the primary risk is the quality of the resulting code. When transitioning from one language to another, the code can have the syntax of the new language but none of the structures or new features. Many source code translators are available to support the transport of code from one language or operating system to another. Source code translators may not solve this problem, since line-by-line translators don't take advantage of target language semantics, constructs, etc., often resulting in code known as "C-TRAN" - C syntax on FORTRAN structures.
While the initial legacy code was of adequate quality, this does not guarantee that the resulting code will have the same quality. If the quality is not adequate, code may have to be improved. If 20 - 30% of the translated code must be changed to improve its quality or to meet standards, the code should not be used and those functions or components should be re-engineered using another track. Quantifying the functionality of the legacy system will provide a basis for estimating how complete the new system is during development and provide estimations as to when the target system will be complete. One method being tested by the SATC and other companies to track progress in re-engineering uses function points as a measure of functionality. In this application, using the basic function point counting method described by Albrecht, an approximate estimate can be gleaned as to the type and count of tasks. This can be used as a starting point in comparing progress in transferring functionality between the original system and the target system. Tools are available to count function points from COBOL code and the SATC is working to develop a tool for FORTRAN and C. In evaluating the progress of the translation, one measure might be the rate at which functionality, approximated through function points, is moving from the existing system to the target system. The ratio of code translated can also serve as a measure of progress.

Once the re-engineering is complete, it is important to verify that the functionality is retained in the new system, as well as the quality of the code has improved, hence implying improved maintainability. There is no simple method to ensure functionality has transitioned between systems. The most commonly used method involves running test cases on the original system and then repeating the tests on the completed target system. It is also important to make sure that documentation is up to date. Documentation that originally applied presumably still does; this translated code represents the most stable part of the system. An added advantage
of the translation track is that maintenance personnel will still understand the program flow and existing documentation since it remains basically unchanged. This will aid them in becoming effective in the new language and system.

5.18.1.2 COTS Track Hybrid Re-engineering

In the COTS track of Hybrid re-engineering, shown in Figure 5.6, requirements and functions must be identified that can feasibly be implemented using COTS.

![Figure 5.6. COTS Track Hybrid Reengineering](image)

After applying the techniques of Reverse Re-engineering to identify the requirements, it is important to separate those requirements that must be contained in the target system from those requirements that users want in the new system because they have become habits or are comfortable with those features. This separation of requirements into "necessary" and "nice" is critical to COTS selection.

The advantage to using COTS is the decreased development time and increased reliability. Testing is still needed during integration, but less than for code developed from scratch. The disadvantage can be the number of requirements...
satisfied by the package. For example, if in the existing system a specific field has 10 characters and the COTS package only allows 8, is this acceptable? Was the 10 character field arbitrary or does it represent a "hidden" business requirement that should not be altered? In addition, users are often resistant to change, even ones that have little impact, such as different icons or keystrokes to call a function. This must be taken into account for the package to have comparable usability and functionality.

Although the use of COTS software decreases the development time and increases the reliability, COTS also introduce additional risks. A major risk is that the package will not perform as anticipated or advertised, that it is unreliable, immature or incomplete. The package may also undergo frequent manufacturer version enhancements requiring constant upgrading. In the worse case, changes may alter or remove functions needed for the system. The COTS may require modifications or supplementation to match the requirements, causing increases in schedule and decreasing reliability. In addition, the use of a COTS may limit further enhancements to the system, since changes in the COTS provided functions may not be possible due to legal issues.

The stability of the vendor should also be part of the evaluation process since it may be necessary for them to make later required changes. An additional cost may be incurred due to the unfamiliarity with the COTS. Simple changes to usability, such as new icons, will require additional training time.

Some metrics are applicable to assist in decreasing the risks associated with the selection of COTS packages. It is important to first identify what percentage of the desired requirements the package totally meets, and which requirements the
package partially meets. This information can then be used to determine how much rework or supplementation the package requires in order to totally fulfill the system requirements. Modification or supplementation will impact the schedule and budget, and may well impact maintainability and reliability. After all of these evaluations are complete, the cost to develop from scratch should also be estimated, including testing time, and compared to the total costs of the COTS.

In this track, once the COTS have been implemented, the functionality of the existing system must be compared to the functionality of the target system. The process of comparison must be based on testing, as was done for translated code functions.

5.18.1.3 Custom Track Hybrid Re-engineering

The Custom track of Hybrid Re-engineering, shown in Figure 5.7, is similar to traditional reengineering since new code is derived from the existing legacy system.

Figure 5.7. Custom Track Hybrid Reengineering

In this track, reverse engineering is first performed. Those functions that are not satisfied by COTS packages or through translated code must be identified, and their requirements and design extracted. Forward engineering is then performed.
This begins with requirements analysis, with the objective of identifying requirements that are not needed. The process is then similar to any development process, beginning with developing a new design, with object-oriented structure if desired, then implementing the code and doing comprehensive testing.

The advantages to the custom track is that the resulting code should meet its requirements exactly. The developed code should be of high quality and well structured, requiring little corrective maintenance. The disadvantages are similar to standard software development, in that the code might not be reliable, requiring additional testing, and that the development/testing process may exceed time and cost budgets. Custom code has the same inherent risks that any code has, to quality, reliability, and schedule.

Since most of the functions of the legacy system identified as unique to the system will be done with custom code, the risk is that one of the unique features will not be identified and hence the functionality of the new system will be incomplete. Features identified as critical to the system that are accomplished with custom code will require extensive testing.

Metrics for this track are a combination of both process and product metrics. Prior to reverse engineering, the quality of the existing system should be evaluated for later comparison to the target system (as discussed previously). Effort expended should be tracked to assist in the evaluation of the cost to re-engineer. This can also be used to approximate schedule completion using the estimate that 60% of the time is in reverse engineering. Once requirements are respecified, their quality can be evaluated to determine testability. Also discussed previously was the use of function points as a means of calculating the rate of functionality transfer to the target system. Code analysis tools can be used to evaluate the quality of the code as
it is being developed and identify the risks. In testing, discrepancy or error rates help in evaluating reliability. In this track, both the functionality and the quality are compared between the existing system and the target system.

5.18.1.4 Hybrid Re-engineering Approach

Hybrid re-engineering requires an approach similar to traditional re-engineering, but with additional considerations. When starting to re-engineer, initial justifications for re-engineering such as costs and quality are developed and expectations, such as return on investment, are stated. An analysis of the legacy system should be done to determine the feasibility of hybrid reengineering.

The analysis of the legacy system should provide a guideline in identifying optimal strategies (translation, COTS, etc.), and projecting the cost of the target system. Once the decision on using a hybrid re-engineering approach is made, additional analysis is needed. The first step in a hybrid re-engineering approach is to investigate the requirements and constraints of the development. These factors include setting a time table for reverse and forward engineering. Time must be built in to investigate available COTS, including hands-on testing of the COTS. While forward engineering development time should decrease with the use of COTS and the translation of code, additional time will be needed for testing the integration and interface of the products of the three tracks. Budget constraints must also be considered; how much can be spent on COTS that provide required features versus those that provide desired features. Management mandated and organization needs must also be identified. As the three tracks are developed, tradeoffs will be necessary so it is important to prioritize requirements.
The next step is to do an in-depth analysis of the legacy system, focusing on functionalities and code segments suitable for each of the three tracks (Translation, Custom, COTS). In generic reengineering, an analysis of the existing system is usually done to provide an evaluation of the quality of the existing system and maintenance costs. This information is used to justify the costs and improvements at the conclusion of the re-engineering effort. While these reasons are still relevant in hybrid re-engineering, additional features of the legacy system must now be investigated. During the assessment of the legacy system, sections and functionalities must be identified. These must be further assessed to determine what documentation is available to identify the required features versus what is no longer needed or what users have become accustomed to. Code sections must be ordered by the cost of maintenance, and the quality of the current structure. Functions that are unique to this project must be identified. All of these components will be used to identify which hybrid re-engineering track will be applied to the code section.

Once the code has been divided into the development tracks, each track will proceed independently as discussed. The schedule for track completion will differ based on tasks. As the tracks conclude various tasks, the merging of the final products can begin. For example, the custom glue can be started once the COTS have been selected. Training on these packages can also begin.

Once the system is complete and all tracks merged, two tasks remain: testing and justification. First, comprehensive System and Integration testing must be performed to ensure all components work together as a cohesive unit and to ensure all functionality of the existing system was transferred to the new system. Second, justification for the re-engineering is usually required – do the benefits gained
justify the cost. Some anticipated benefits, such as improved maintenance and operational costs, can only be demonstrated indirectly through the improved quality. Improved quality can initially be demonstrated by a metric analysis of the legacy system compared to the new system. As the new system is put into operation, additional metrics can be used to verify the improvements.

5.18.1.5 Hybrid Re-engineering Risks

All software development has inherent risks to schedule and cost. Hybrid re-engineering, as a software development methodology, is also susceptible to them. Hybrid re-engineering, because of its composition of the three diverse development tracks, is subject to all of the risks that were discussed within each track description. Also, Hybrid re-engineering as a unique software reengineering methodology has additional risks to functionality and quality; the functionality of the existing system must be preserved in the new system, and the quality must improve, implying a decrease in operational and maintenance costs. With all of the risks in Hybrid re-engineering, why bother, why not just treat it as a new software development effort and omit the reengineering all together? Because of the benefits.

5.18.1.6 Hybrid Re-engineering Benefits

In general, re-engineering is performed as opposed to building a new system because of the invisible business application procedures and logic that are built into the software. These processes might be deeply embedded in business procedures as simple as a field length or as complicated as a mathematical algorithm; the only source of this information is in the legacy code. A second justification for re-engineering versus building is the development and maintenance costs of the legacy system; the time spent developing logic and
components should not be wasted. In re-engineering, the existing system is re-implemented and instilled with good software development methodologies, properties, and new technology while maintaining existing functionality. Reliability and maintainability are also improved.

Hybrid re-engineering has the additional benefits of a reduced development schedule, hence reduced costs. The development schedule is shortened first by minimizing the amount of reverse engineering (recall, reverse engineering is 60% of the effort). The translation track uses minimal reverse engineering time since work is done in the lowest level. The use of COTS decreases the forward engineering development and test time and thus the costs. The use of properly selected COTS also increases the reliability since these packages have been extensively tested.

5.18.1.7 Hybrid Re-engineering Metrics

Metrics, when properly applied, provide managers information about their project to help mitigate risks. It is logical therefore, to discuss some of the re-engineering phases where metrics provide valuable information. Previously we have identified metrics applicable for each track in hybrid re-engineering. In this section, we will discuss metrics applicable to the entire project, not just one track. Metrics provide information on the quality, functionality, and on track selection, a prime area of risk.

At the start of the re-engineering effort, the legacy system must be quantified. There are two objectives:
5.18.1.7.1 Identify the amount of functionality and
5.18.1.7.2 The quality of the existing system.

By quantifying the functionality, scheduling estimates are more accurate; during
development, completion can be estimated by the percentage of functionality
transferred to the new system. Functionality is also important at the conclusion of
the project, measuring how much functionality is contained within the new system.
The SATC and others are working with function points as a means of estimating
functionality. Function points are comparable across languages, and time estimates
based on function points are available. For COTS packages, functionality might be
measured by the number of requirements satisfied.

Quality is harder to measure and few software developers agree totally on the
appropriate metrics. The SATC has a group of metrics it applies to projects to
evaluate the quality. These metrics evaluate the project at the module level
(procedure, function, class or method). The size is measured by counting the
number of executable statements. The readability is measured by the comment
percentage.

The complexity is measured by the cyclomatic complexity (McCabe). The
coupling is measured by the calling complexity (fan in/fan out). One final measure
of quality is the reliability, the number of errors found, and the projected number
of errors remaining. These metrics can be used for both the translation track and
the custom code track. When the components are combined, the numbers of errors
found and the projected number of errors remaining can be applied to the whole
system.
5.19 SUMMARY

The number of large systems being built from scratch is diminishing, while the number of legacy systems in use is very high. Rapid changes in the computer industry continually introduce new hardware and software, making older systems obsolescent and difficult to maintain. Businesses do not want to re-develop from scratch; too many business decisions are built into the legacy systems, hence reengineering.

But project development is always short on time and money, making it necessary to look at alternatives. The use of COTS packages is seen as a way to increase reliability while decreasing development and test time. Translation of code is a means of decreasing time and cost. This results in a combination of development methods into a form of hybrid re-engineering. Although any type of re-engineering has associated risks, metrics are available to help identify and mitigate the risks, leading to successful software development.