Software Re-engineering

Prepared By:

Dr. Linda H. Rosenberg

Engineering Section head

Software Assurance Technology Center

Unisys Federal Systems

301-286-0087

Linda.Rosenberg@gsfc.nasa.gov

Accepted By:

Lawrence E. Hyatt

Manager, Software Assurance Technology Center

System Reliability and Safety Office

Goddard Space Flight Center, NASA

301-286-7475

Larry.Hyatt@gsfc.nasa.gov

Preface

The essence of software re-engineering is to improve or transform existing software so that it can be understand, controlled, and used anew. The need for software re-engineering has increased greatly, as heritage software systems have become obsolescent in terms of their architecture, the platforms on which they run, and their suitability and stability to support evolution to support changing needs. Software re-engineering is important for recovering and reusing existing software assets, putting high software maintenance costs under control, and establishing a base for future software evolution. The growth in cost and importance of software to NASA, and the aging of many of the Agency's important software systems, has necessitated software re-engineering efforts.

This technical report is designed to give the reader an overview of the concepts, approaches and risks of re-engineering. It is intended to serve as a basis for understanding software re-engineering technology. The information is primarily compiled from experts referenced at the end of the report, modified by approaches taken by NASA Projects. Section 8, however is new. It

discusses Hybrid Re-engineering, an approach used by some NASA projects to combine the use of Commercial-Off-The-Shelf (COTS) software with new software development. The technical reports concludes with some industry lessons learned.

1. Introduction

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. 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 Usually requirements, design and code documentation is no longer available, or is very out of date, so it is unclear what functions are to be moved.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 paper 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 re-engineering, Hybrid re-engineering, is defined and discussed. The final section looks at experience reports from industry: what worked and what didn't.

2. 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 reverse engineering, redocumentation, restructuring, translation, and 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.

3. Re-engineering Objectives

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:

- Preparation for functional enhancement
- Improve maintainability
- Migration
- Improve reliability

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.

4. Re-engineering Concepts

4.1 Software Development Levels of Abstraction

Levels of Abstraction, Figure 1, that underlie the software development process also underlie the re-engineering 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).

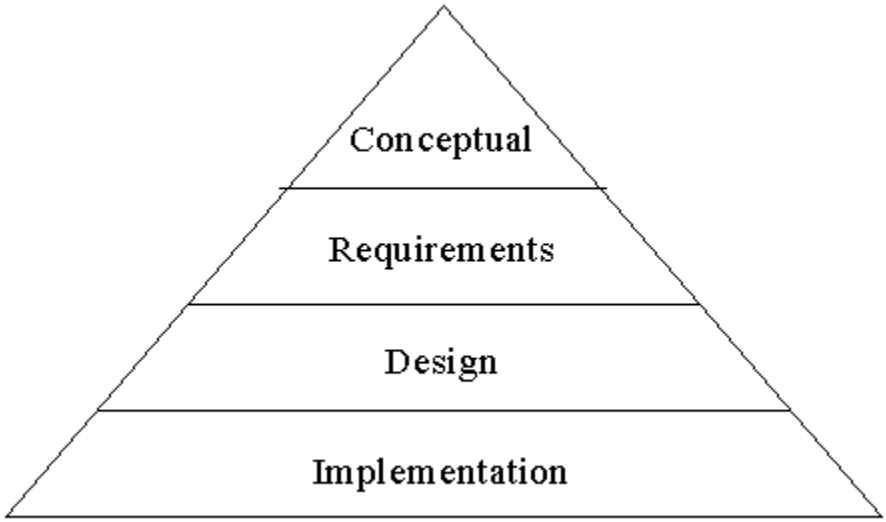

Figure 1: Levels of Abstraction

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.

4.2 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 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.

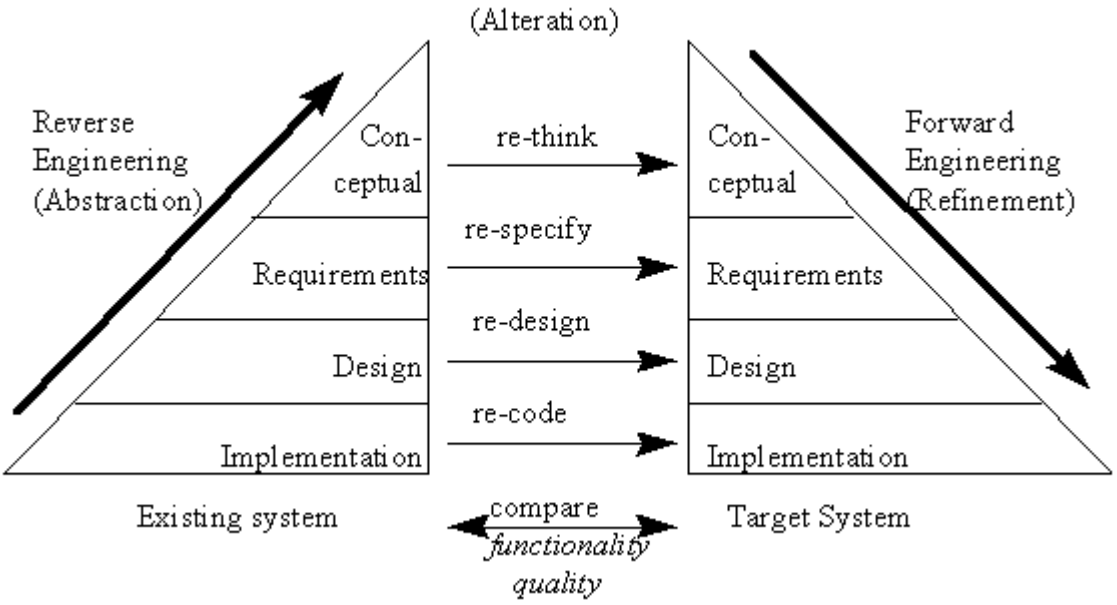

Figure 2: General Model for Software Re-engineering

The model in Figure 2 applies three principles of re-engineering: abstraction, alteration, and refinement. Abstraction is a gradual increase in the abstract level of system. 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.

4.3 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. In reverse engineering, the requirements and the essential design, structure and content of the legacy system must be recaptured. In addition to capturing

technical relationships and interactions, information and rules about the business application and process that have proved useful in running the business must also be retrieved. This involves extracting design artifacts and building or synthesizing abstractions that are less implementation dependent. The key objectives in reverse engineering are to generate alternative views, recover lost information, detect side effects, synthesize higher abstractions, and facilitate reuse. The effectiveness of this process will affect the success of the reengineering project. Reverse engineering does not involve changes to the system or creating a new system, it is the process of examination without changing its overall functionality.

The reverse engineering process, shown in Figure 3, begins by extracting the requirements and detailed design information from the source code and existing documents. A requirements document is created and a high level design abstraction is extracted and expressed using data-flow and control-flow diagrams. The recovered design is reviewed for consistency and correctness.

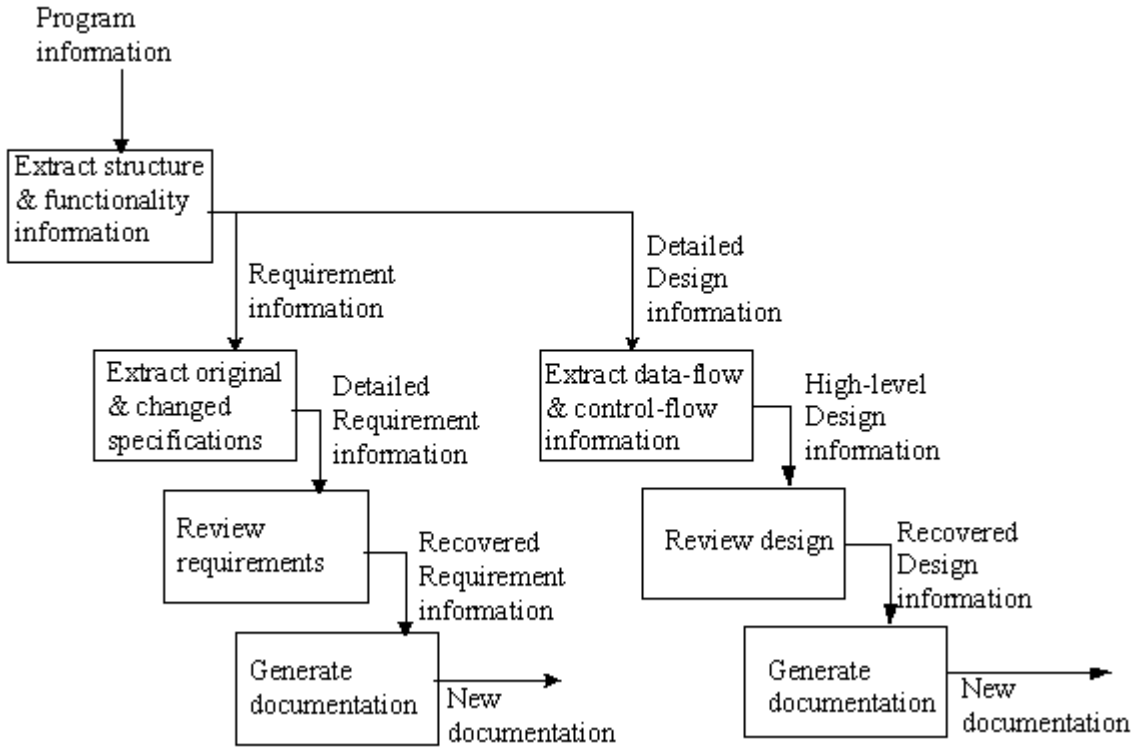

Figure 3: Reverse Engineering Procedure

Slicing, or code isolation, is a method for pulling out from a program those parts that are concerned with a specified behavior of interest. It is a means of locating the minimum amount of data and code that is required to derive a desired data item at a particular point in the program. Conceptually, slicing is easy - accrue all data definitions and control flow which contribute to the desired data element values at the slice points and discard anything that does not contribute to those values. The objective of slicing is to compute the minimal subset of code required for a function. However, in reality slicing is very difficult.

There are three approaches to slicing: conditional-based, forward, and backward.

Conditional-based slicing utilizes the concept that business functions are structured along conditional tests. This slicing method is potentially useful to identify areas of the program reachable under globally specified conditions since the user specifies logical express and operationally slicing range (where to start and end slicing). All reachable program statements along control flow paths for which given expression is true are included in the slice segment. Forward slicing uses functions that normally process input values. The objective in this method is to find all areas of code that depend on the values of input variables. Ripple effect analysis is applied; given a variable and slicing range, statements in that range that can be potentially affected by the variable are defined. The statements can be affected in terms of data flow, control dependency, or recursive process; variables on the left and right of the statement are included. In backward slicing, the process starts with the results of the existing program. Areas of code that contribute to values of output variables are a part of the slice. Given a variable and range, this method of slicing returns statements that can potentially affect the value of variable; variable only on right side of the equation are used. Tools are available to facilitate slicing, but the process is still difficult and time consuming.

4.4. Forward Engineering

The new target system is created by moving downward through the levels of abstraction, a gradual decrease in the abstraction level of system representation by successive replacement of existing system information with more detailed information. This downward movement is actually forward movement through the standard software development process, hence, forward engineering. Forward engineering moves from high level abstractions and logical implementation independent designs to the physical implementation of the system. A sequence from requirements through design to implementation is followed. In this process, the risks are due to the degree and preparation during reverse engineering. Projects are exposed to more risk with the alteration or addition of new requirements.

5. Re-engineering Approaches

There are three different approaches to software re-engineering. The approaches differ in the amount and rate of replacement of the existing system with the target system. Each approach has its own benefits and risks.

5.1 Big Bang Approach

The "Big Bang" approach, also known as the "Lump Sum" approach, replaces the entire system at one time, as shown in Figure 4. This approach is often used by projects that need to solve an immediate problem, such as migration to a different system architecture.

Figure 4: Big Bang Approach

The advantage to this approach is that the system is brought into a new environment all at once. No interfaces between old and new components must be developed, no mingled environments must be operated and maintained. . The disadvantages with this approach is the result tends to be monolithic projects that may not always be suitable. For large systems, this approach may consume too many resources or require large amounts of time before the target system is produced. The risk with this approach is high, the system must be functionally intact and work in parallel with the old system to assure functionality. This parallel operation may be difficult and expensive to do. A major difficulty is change control; between the time the new system is started and finished, many changes are likely to be made to the old system, which have to be reflected in the new system. It is a lot of work to stay current and not to lose a capability that has been put in the old system. That is, what is being reengineered is likely to change.

5.2 Incremental Approach

The "Incremental" approach to re-engineering is also know as "Phase-out". In this approach, shown in Figure 5, system sections are re-engineered and added incrementally as new versions of the system are needed to satisfy new goals. The project is broken into re-engineering sections based on the existing system's sections.

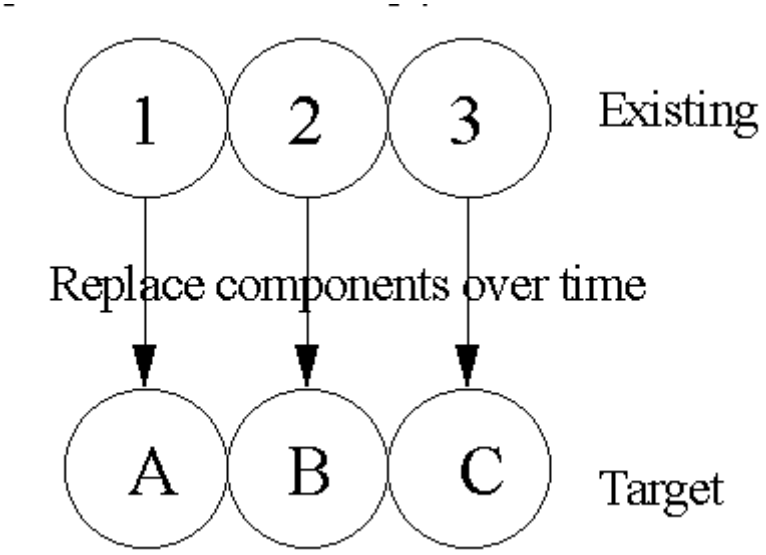

Figure 5: Incremental/Phase-out Re-engineering Approach

The advantages to this approach are that the components of the system are produced faster and it is easier to trace errors since the new components are clearly identified. Since interim versions are released, the customer can see progress and quickly identify lost functionality. A benefit is that change to the old system can be easier dealt with, since changes to components that are not being re-engineered have no impact on the current component. A disadvantage to the Incremental approach is that the system takes longer to complete with multiple interim version that require careful configuration control. Another disadvantage is that the entire structure of the system cannot be altered, only the structure within the specific component sections being re-

engineered. This requires careful identification of components in the existing system and extensive planning of the structure of the target system. This approach has a lower risk than the Big Bang because as each component starts re-engineering, the risks for that portion of the code can be identified and monitored.

5.3 Evolutionary Approach

In the "Evolutionary" approach, as in the Incremental approach, sections of the original system are replaced with newly re-engineered system sections. In this approach however, the sections are chosen based on their functionality, not on the structure of the existing system. The target system is built using functionally cohesive sections as needed. The Evolutionary approach allows developers to focus re-engineering efforts on identifying functional objects regardless of where the tasks reside in the current system. As shown in Figure 6, components of the current system are broken by functions and re-engineered into new components.

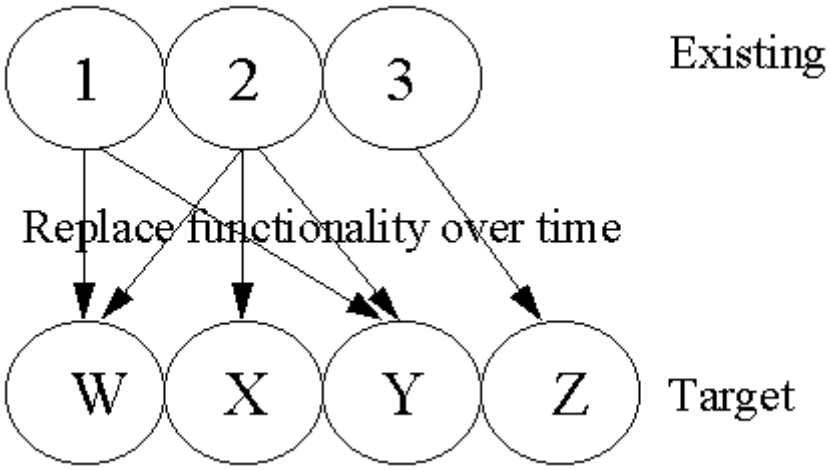

Figure 6: Evolutionary Re-engineering Approach

The advantages of Evolutionary re-engineering are the resulting modular design and the reduced scope for a single component. This approach works well when converting to object-oriented technology. One disadvantage is that similar functions must be identified throughout the existing system then refined as a single functional unit. There may also be interface problems and response time degradation since functional sections of the original system are being re-engineered instead of architectural sections.

6. Re-engineering Phases and Tasks

There is a core process that every organization should follow when re-engineering. Re-engineering 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 re-engineering, and concluding with the transition to the new target system. These five re-engineering development phases are: 1) Re-engineering team formation; 2) Project feasibility analysis; 3) Analysis and planning; 4) Re-engineering implementation; and 5) Transition and testing.

6.1 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.

6.2 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 re-engineering must be compared to the expected cost savings and the increase in value of the system.

6.3 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.

6.4 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 in Figure 2, 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.

6.5 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.

7. Re-engineering Risks

Although re-engineering is often used as a means to mitigate risks and reduce costs of operating and maintaining the legacy software, re-engineering is not without risks. Early risk identification assists program and project managers in preparing for estimation and evaluation of software re-engineering risks and provides a realistic framework for expectations. Risk identification is essential for effective risk assessment, risk analysis and risk management. These potential risks are discussed throughout this report, and consolidated in Table 1 by risk areas.

Risk Area	Risk
Process	Extremely high manual re-engineering costs
	Cost benefits not realized in required time frame
	Re-engineering effort drifts
	Lack of management commitment to ongoing re-engineering solution
	Subsystems chosen incorrectly for approach
	Inadequate configuration management
	Lack of quality assurance
	Lack of metrics program
	Re-engineering with no local application experts available
	Re-engineered system does not perform adequately
	Masses of expensive documentation is produced
	Re-engineering technology inadequate to accomplish re-engineering goals
	Legacy functionality becomes obsolete prior to re-engineering completion
Reverse Engineering	Language not designed to express abstract information necessary for requirement and design specifications
	Difficult to capture much design and few requirements from code

	Objects captured incomplete or incorrect
	Existing business knowledge embedded in source code is lost
	Recovered information is not useful or not used
Forward Engineering	New requirements and functionality added
	Captured objects do not integrate to new system
	Difficulty in migrating existing data
	Degree of preparation and reverse engineering insufficient
Personnel	Personnel without appropriate skill level and experience in re-engineering
	Programmers performing less effectively to make an unpopular re-engineering project look less effective
Tools	Dependence on tools that do not perform as advertised
	Availability - new and immature, often incomplete functionality
	Maturity - stability of tool vendor and tool quality
Strategy	Premature commitment to a re-engineering solution for an entire system
	Failure to have a long-term vision with interim goals
	Vulnerable, unrealistic goals
	Approach chosen does not meet company goals, budget or schedule
	No plan for using re-engineering tools

---

8. 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 7, an adaptation of the General Model for Software Re-engineering shown in Figure 2.

In Figure 7, 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.

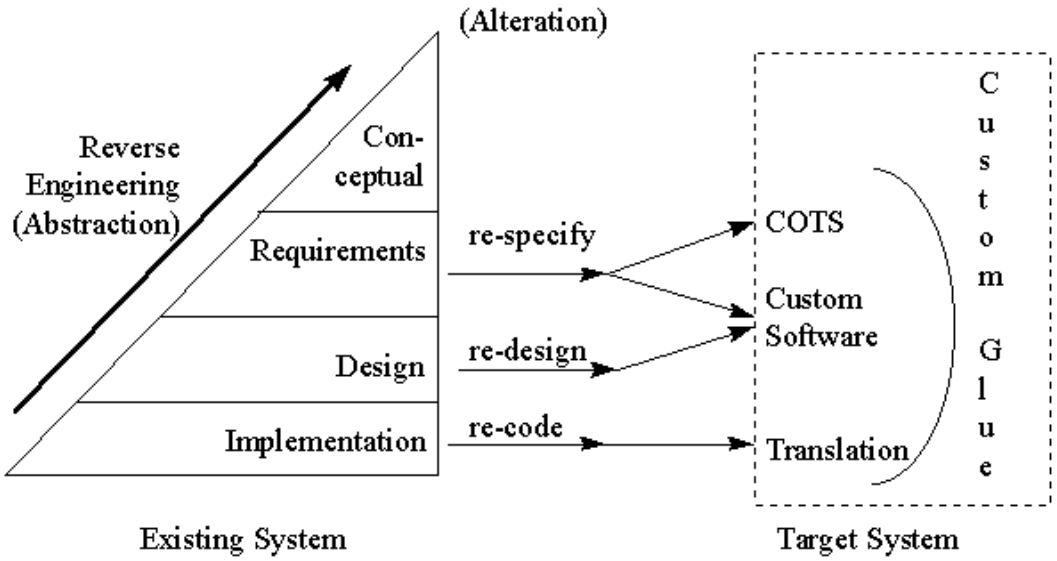

Figure 7: Hybrid Re-engineering Tracks

Re-engineering as a development methodology has inherent risks as shown in Table 1, 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 re-engineering, metrics can support the justification for decisions on track selection for different software functions and components.

8.1 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.

8.1.1 Translation Track Hybrid Re-engineering

Figure 9 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).

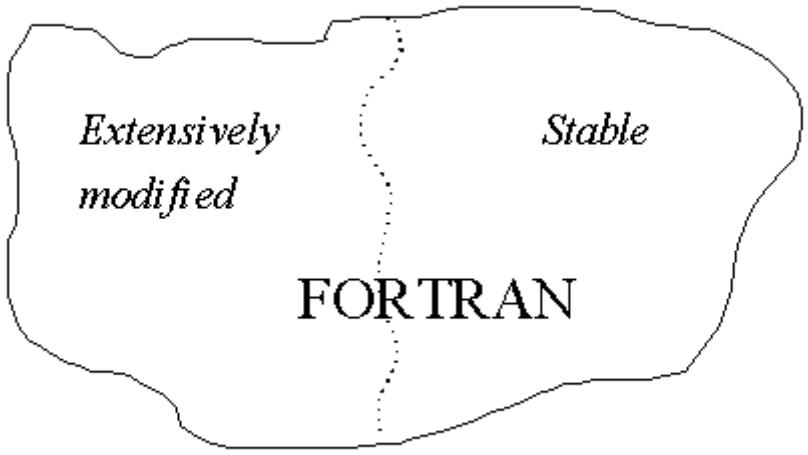

Figure 8: Current Software System

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 under-gone 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, shown in Figure 4, 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.

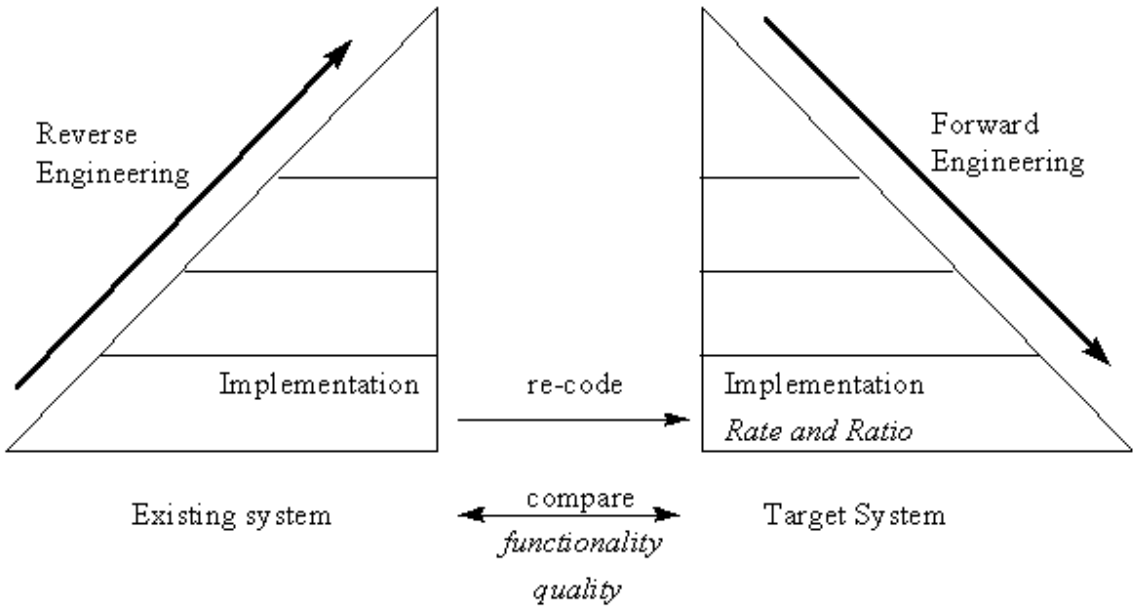

Figure 9: Translation Track Hybrid Re-engineering

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 know 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 functionally 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.

8.1.2 COTS Track Hybrid Re-engineering

In the COTS track of Hybrid re-engineering, shown in Figure 10, requirements and functions must be identified that can feasibly be implemented using COTS.

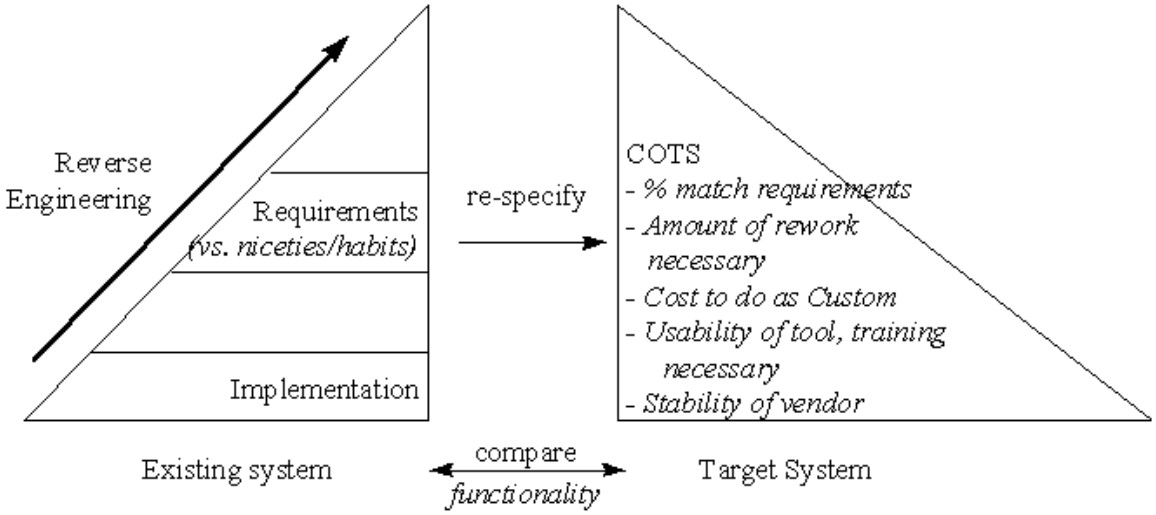

Figure 10: COTS Track Hybrid Re-engineering

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.

8.1.3 Custom Track Hybrid Re-engineering

The Custom track of Hybrid Re-engineering, shown in Figure 11, is similar to traditional re-engineering since new code is derived from the existing legacy system.

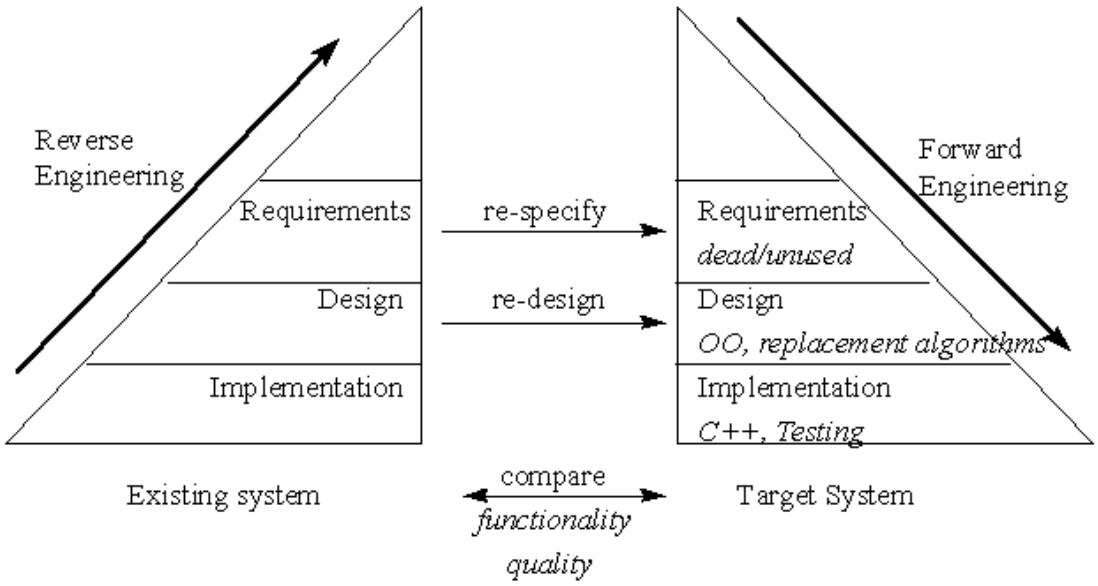

Figure 11: Custom Track Hybrid Re-engineering

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 re-specified, 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.

8.2 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 re-engineering. 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 re-engineering, 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.

8.3 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 re-engineering 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 re-engineering all together? Because of the benefits.

8.4 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 (Figure 8). 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.

8.5 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 areas of risk.

At the start of the re-engineering effort, the legacy system must be quantified. There are two objectives: identify the amount of functionality and 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). Although there are many other metrics available, these have been successfully applied to many projects at GSFC NASA. 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.

9. Re-engineering Industry Applications

9.1 Results

9.1.1 Effort and Productivity

A re-engineering case study conducted by the National Institute of Standards and Technology (NIST) investigated moving from an existing Assembly language and unstructured COBOL system to a structured COBOL system. The re-engineering methodology was broken into five steps and the effort for each was tracked, results are shown in Table 2.

Steps	Percentage of Total Effort
Baseline current system	20 %
Extract /analyze data, code functionality, documentation	43 %
Produce documentation	4 %
Generate new code	26 %
Execute and test code	7 %

Table 2: Effort per Activity

The data in Table 2 can be summarized to approximately 60% of the effort in reverse engineering and approximately 40% of the effort expended in forward engineering.

This case study also tabulated the following productivity measures. Using a conversion factor for function points to C source statements, additional measures are included.

164 function points per staff year (~ 21,000 C source statements)

11.68 staff hours per function point (~ 128 C source statements)

1,516 executable statements per staff year

6,387 COBOL statements per staff year

9.1.2 Benefits and Costs

Sneed has been working to determine the cost of re-engineering in terms of value added based on the risk factor, and in terms of effort expended. When calculating the benefit of replacing a technically obsolete system that must be replaced, Equation 1 below is applied.

(Old value - (re-engineering cost * re-engineering risk))

+ (new value - (development cost * development risk))

Equation 1: Benefit calculation

When a system is being re-engineered because of severe technical problems with the existing system, the maintenance costs become a factor as shown in Equation 2.

(Old maintenance cost - re-engineering main cost)

+ (old value - (re-engineering cost * re-engineering risk))

- (new value - (development cost * development risk))

Equation 2: Benefit calculation including maintenance costs

Table 3 below specifies the risk factor used in Equations 1 and 2, based on the probability of failure.

Risk rating	Probability of failure	Risk factor
Extremely high	0.99 - 0.81	3
very high	0.80 - 0.61	2.5

High	0.60 - 0.50	2
Moderate	0.49 - 0.25	1.5
Low	0.24 - 0.10	1.25
None	0.09 - 0.01	1

Table 3: Risk Factor for Benefit calculations

One factor in calculating the cost and benefit of re-engineering is the amount of effort necessary. One method developed by Sneed breaks the system into component parts, weights the effort for each unit, then sums the results for the total effort needed. These calculations are shown below.

E1 = (number programs * program effort)
    * (1 + (number IOs/number programs + number of IOs))
E2 = (number subroutines * subroutine effort)
    * (1 + (number CALLs/number subroutines + number CALLs))
E3 = (number jobs * job effort)
    * (1 + (number files/number jobs + number files))
E4 = (number files * file effort)
    * (1 + (number IOs/number files + number IOs))
E5 = (number copys * copy effort)
    * (1 + (number ref/number copys + number ref))
E6 = (number panels * panel effort)
    * (1 + (number fields/number panels + number fields))
E = (E1 + E2 + E3 + E4 + E5 + E6) * 2 (test effort)

Figure 12: Total Re-engineering Effort Calculation

9.1.3 Complexity and Reuse

One general objective when re-engineering is to decrease maintenance costs. Since complexity has been closely tied to the cost of maintenance (higher complexity increases the cost of maintenance), determining the effect of restructuring vs. re-engineering is one method for showing cost savings. Sneed compiled the statistics shown in Table 4.

Metric	**Unstructured**	**Restructured**	**Re-engineered**
Module complexity	0.86	0.33	0.38

Graph complexity	29	25	19
Data complexity	1.87	1.83	1.66
Difficulty degree	0.033	0.031	0.044
Test complexity	0.409	0.360	0.354

Table 4: Affects of Re-structuring and Re-engineering on Complexity

Sneed also looked at specific elementary program measurements shown in Table 5.

Measure	**Unstructured**	**Re-structured**	**Re-engineered**
External inputs	19	18	13
Procedures	1	11	10
External outputs	25	23	19
Interfaces	20	20	16
Predicates	8	18	15
Total Inputs	61	78	76
Total Outputs	52	52	47
Data Used	101	129	129
GOTOs	13	1	0
PERFORMs	0	16	18
UNTIL loops	0	2	2
IFs	14	16	11
ONs	3	4	4
Branches	44	61	48
paths	18	22	17
LOC	377	546	548
Stmt	137	163	160
Stmt types	16	18	20

Data ref	348	426	341
Files	7	7	7

Table 5: Program Measurements

In the case study conducted by NIST discussed above, similar program metrics were calculated before and after re-engineering. The results were normalized to a base of 100 executable lines of code and are shown in Table 6.

Metric	Before Re-engineering	After Re-engineering
Executable stmts	3116	4062
Number of Programs	14	14
Total decision count	710	1376
Decision density	22.74	33.89
Total function count	537	1407
Total I/O count	384	335
Entry/exit ratio	0.5	0.66
GOTO density	0.25	6.67
NOT clauses	159	466
Functions/procedures	38.36	100.5
Function density	17.2	34.66
Total number of files	46	57
Total number of calls	42	8
Comments density	118.87	318.04

Table 6: NIST Program metrics

In a study done by Basili, in addition to complexity metrics, Halstead Software Science metrics of Volume and Length were used. The reuse frequency was calculated by the number of static calls addressed to a component compared to number of call address to class of components assumed to be reusable. The purpose of this study was to identify components within an acceptable range, therefore candidates for translation from one language (or platform) to another with minimal effort. Table 7 shows his results.

Measure	Minimum	Maximum
Volume	2,000	10,000
Complexity	5	15
Regularity	0.70	1.3
Reuse Frequency	0.30	

Table 7: Measures for Reuse ==> Translation

The results from Tables 5, 6 and 7 can be summarized as shown in Table 8, identifying the expected increase or decrease in the metrics when programs are restructured or re-engineered.

Metric/measure	Movement
Module complexity	Decrease
Graph complexity	Decrease
Data complexity	Decrease
Test complexity	Decrease
Procedures	Increase
Interfaces	Decrease
Inputs	Increase
Outputs	Decrease
Branching	Increase

Table 8: Metric Summary

9.2 Industry Lessons Learned

As companies apply re-engineering techniques, valuable lessons can be learned. Some general lessons learned are below.

- **Re-engineering is not a panacea and can never compensate for lack of planning, do not expect miracles, many small gains can eventually add up to one large gain.**
- **Identify motivations and what is to be achieved by re-engineering.**
- **Direct, visible benefits to the customer are vital for continued support.**
- **Management commitment is crucial, learning curves require extra time, extra time causes slippage.**
- **Cost of planning, training, execution, testing and internal marketing are real engineering costs.**
- **Re-engineering requires a highly trained staff that has experience in the current and target system, the automated tools, and the specific programming languages.**
- **It is critical that the application system experts be involved throughout the re-engineering process. They are essential for design recovery to identify hidden history.**
- **Tools should be available early in the process, easy to learn and use, and compatible with the requirements of the particular enterprise.**
- **Be ready for CASE technology before you get it, find a tool that fits and apply it only after processes are well defined.**
- **Evaluate the system functionality with the intent of discovering what is worth retaining for future use and what is not.**
- **While design recovery is difficult, time-consuming, and essentially a manual process, it is vital for recovering lost information and information transfer.**
- **Evaluate the code, documentation, maintenance history and appropriate metrics to determine the current condition of the system.**

10. Conclusion

As the software industry moves to a new millennium, many new software design methodologies are developed, improving software reusability and maintainability, and

decreasing development and maintenance time. But most companies have legacy systems that are out of date and costly to maintain. These system cannot just be replaced with new systems, they contain corporate information and implied decisions that would be lost. They also are an investment, and were too costly to develop and evolve just to discard. For these purposes, re-engineering becomes a useful tool to convert old, obsolete systems to more efficient, streamlined systems. But project development is always short on time and money, making the need to look at alternatives necessary. 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 has resulted in a combination of the development methods into a form of hybrid re-engineering.

Re-engineering is a structured discipline of software development and different approaches and specific phases and tasks. However, extensive work is still needed. Although it is noted that configuration management is important, there are no methods for incorporating it. There also is not a methodology for evaluating the quality of the new systems or COTS. Metrics are also needed throughout the re-engineering process, but it is not clear exactly how concepts could be measured, and that the metrics would in fact evaluate what they are proposed to quantify.

This paper is a summary of existing strategies and techniques in re-engineering, to serve as a basis for future work.

References:

Albrecht, A., Gaffney, J. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Transactions on Software Engineering*, 11/83.

Adolph, W. Stephen, "Cash Cow in the Tar Pit: Reengineering a Legacy System", IEEE Software, 5/96.

Arnold, Robert S., "Software Restructuring", Proceedings IEEE, Vol 77, No 4, 4/89.

Byrne, Eric J., "A Conceptual Foundation for Software Re-engineering", Conference on Software Maintenance, 1992.

Byrne, Eric J., "A Software Re-engineering Process Model", 2nd International Conference on System Integration, 1992.

Byrne, Eric J., "Software Reverse Engineering: A Case Study", Software- Practice and Experience, 12/91.

Caldiera, Gianluigi, and Basili, Victor R., "Identifying and Quantifying Reusable Software Components", IEEE Computer, 2/91.

Chikofsky, Elliot J., and Cross, James H., "Reverse Engineering and Design recovery: A Taxonomy", IEEE Software, 1/90.

DISA, "A Software Reengineering risks Taxonomy", Software Technology Conference, 4/95.

Federal Software Management Support Center, "Parallel Test and Productivity Evaluation of a Commercially Supplied Cobol Restructuring Tool", Office of Software Development and Information Technology, 8/89

Feiler, Lamia, Samith, "Reengineering as an Engineering Problem: Conceptual Framework and Application to Community Problems", Software Technology Conference, 4/95.

Grady, R., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.

Hyatt, L., Rosenberg, L., "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality", 8th Annual Software Technology Conference, UT, 4/96.

Jones, C., *Applied Software Measurement*, McGraw Hill, Inc., 1991.

Manzella, Mutafelija, "Concept of re-engineering Life Cycle", IEEE, 1992.

Ning, Jim Q., Engberts, Andre, Kozaczynski, W., "Automated Support for Legacy Code Understanding", Communications of the ACM, 5/94.

Olsem, Michael r., "Preparing to Reengineer", Software Technology Support Center, 2/94.

Olsem, Michael R., Sittenauer, "Preparing to Reengineer and STSC Reengineering Projects Tutorial", Software Technology Conference, 4/95.

Rosenberg, L., Hyatt, L., "Developing a Successful Metrics Program", European Space Agency Symposium, 3/96.

Ruhl, Mary K., Gunn, Mary T., "Software Reengineering: A Case Study and Lessons Learned", NIST, 9/91.

Sittenauer, Chris, Olsem, Michael, Balaban, John, "Software Reengineering at STSC", Software Technology Support Center, 2/94.

Sneed, Harry M., "Economics of Software Re-engineering", Journal of Software Maintenance, 9/91.

Sneed, Harry M., "Planning the reengineering of Legacy systems", IEEE Software, 1/95.

Sneed, Harry M., Kaposi, Agnes, "A Study on the Effect of reengineering upon Software Maintainability", Conference on Software Maintenance, 1990.

Software Technology Support Center, "STS Reengineering Technology Report, Vol 1", Hill Air Force Base, UT, 10/95.

Software Technology Support Center, "STS Reengineering Technology Report, Vol 2",

Hill Air Force Base, UT, 10/95.

Wilson, W., Rosenberg, L., Hyatt, L., "Automated Analysis of Requirements Specification", Fourteenth Annual Pacific Northwest Software Quality Conference, Oregon, 10/96.