CHAPTER - II
REVIEW OF LITERATURE

The concept of re-engineering originated in management theory in early nineteenth century. Re-engineering for software systems became popular in the early 1990s and at that time it was not fully understood or valued. In the twenty-first century, re-engineering of software is an effective instrument for organizations striving to operate as effectively and efficiently as possible. Software re-engineering does not exist from centuries as a result of which, complete and independent texts on this subject are not much available. At the same time, being a topic of vital concern, it has received considerable attention from experts and researchers in the computer journals. Companies are launching re-engineering projects radically due to global competition. As one company becomes more productive, others in the field obligatory follow.

Literature has been reviewed through reputed national and international journals (print and E-journals), reports and databases, archives, conference papers, magazines, books, Internet and newspapers etc. Some doctoral researches on software re-engineering are also seen. The main motive of this review is to come across on the perspective articles and academic research outputs.

Software re-engineering is essential for putting high maintenance cost under control, recovering legacy software assets, and building a base for future evolution. Reengineered software can be used as a new one with another lifespan. Re-engineering can be done of every aged system but here in this study re-engineering means re-engineering of software systems. Software Re-engineering is vital to retain costly legacy software and to avoid new software development. There is no universally accepted definition of re-engineering of software as well as no universally accepted spelling for re-engineering. According to Arnold (1994) the most common spellings of re-engineering are ‘reengineering’ and re-engineering. Re-engineering is also known as renovation or reclamation of the software systems. Here our focus is on studying the literature relevant to re-engineering of object-oriented legacy software systems.
Legacy system is a system which we inherit from our ancestors and is valuable to us. According to The Free On-Line Dictionary of Computing (FOLDOC) legacy software system is ‘A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents’. Ian Somerville [53] defines legacy systems as ‘Older software systems that remain vital to an organization’.

Although the object-oriented paradigm is relatively new, there are many object-oriented software legacy systems. The object-oriented paradigm has been adopted in many software industries and in the coming future object-oriented software would exist everywhere. Accommodating with the trend, the software re-engineering for object-oriented systems has been gathering much attentions for reducing development costs and to facilitate the software maintenance. In the past, most of the re-engineering efforts were focused on re-engineering spaghetti code to modular code and non object-oriented code to object-oriented code. Now, with existence of millions of lines of code in object-oriented legacy software systems, re-engineering of object-oriented legacy systems is an active area of research. Early adopters of the object-oriented programming paradigm now face the problem of transforming their object-oriented legacy systems into the more reusable and flexible frameworks and components. The ability to deal with large and poorly documented object-oriented programs definitely require support from tools as well as methodologies Demeyer et al. [33]. Different people have defined component in different ways. According to Qureshi et al. [85] the component is similar to the object concept of object oriented (OO) programming. A component is an independent part of the system having complete functionalities. The authors evaluate object oriented (OO) process model and propose a novel process model for CBD (Component Based Development). Demeyer et al. [34] there is rush towards new trends that is patterns, components, UML, XML and so on, but objects are not bad, objects are pretty good, and must be taken care of them. The concept of object oriented development methodology helps in constructing software systems that are easy to understand and hence easy to re-engineering. Object oriented concepts are capable to reduce the semantic gap between the real world and the programming
languages Mohamed [75]. Many model(s) for object oriented development have been put forth by the various authors, the well known are Rumbaugh et al. [94], Stefik [106], Jacobson et al. [56]. The Unified Modeling Language (UML) is largely used as a design and documentation instrument for object oriented designs. Quatrani [84] and OMG [80] provide a spectrum of notations for representing different aspects of systems which are accepted as standard notations in the industry.

Oscar [81] FAMOOS (ESPRIT Project 21975- Framework-based Approach for Mastering Object-Oriented Software Evolution) project was launched from November 1996 to December 1999. The goal of the project was to develop tools and techniques to rejuvenate object-oriented legacy software systems. The author participated in the research project. According to the author the idea at the start of the project was to convert big object-oriented applications into frameworks that can be easily reconfigured using a variety of different programming techniques.

Mohan et al. [76] proposes to integrate software product line engineering and agile development in which several important practices such as control scope, re-factor selectively, drive product incrementally, and re-factor for reuse to support component re-engineering.

Software re-engineering covers three major areas: (1) understanding software (2) improving software (3) preserving software. Software re-engineering technology has three major phases: (1) reverse engineering (2) transformation of architecture (3) forward engineering.

In design of model(s) of re-engineering, literature on every aspect of re-engineering process has to be studied. The history of re-engineering, phases of re-engineering, tools and techniques of re-engineering, state of the art in re-engineering must be studied. Cost of re-engineering, present cost model(s) available in the literature, re-engineering experience etc has to be analyzed to design complete model(s). Beginning should be from context, meaning and definitions.

In the literature related to the re-engineering, reverse engineering, forward engineering and software engineering fields, there are many conflicting uses in the terminology. To provide consistency, the taxonomy project of the IEEE-CS technical council on software engineering (TCSE) - Committee on reverse
engineering has been working on a unified taxonomy of the field TCSE [109].

Arnold [3] describes software re-engineering as any activity that (1) improve one’s understanding of software, or (2) prepares or improves the software itself, usually for increased software maintainability, reusability, or resolvability.

In this description software includes documentation, graphical pictures and analysis in addition to source code.

Different scientists have different meaning for re-engineering. GUIDE [43] defines re-engineering as “the process of modifying the internal mechanics of a system or program or the data structures of a system or program without changing its functionality”. According to this definition the functionality of the system remains intact. Re-engineering means overhauling the system as it is like a new development, and therefore functionality can be changed.

According to Chikofsky and Cross, re-engineering means “the examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form”. In this definition the meaning is broader as it describes re-engineering as new form of the system. Re-engineering is more than repairing, renovating and improvement as re-engineering means making the system new in the current environment and technology. But this definition does not touch other non-source code items, such as documentation and specifications. Each author defines re-engineering in his/her own way but is valuable and useful. Yet there is no universal accepted definition of re-engineering. Re-engineering technology and terminology is changing fast.

The critical distinction between re-engineering and new software development is the starting point for the development. Rather than start with a written specification, the old system acts as a specification for the new system.

'Re-engineering' is a set of activities that are carried out to re-structure a legacy system to a new system with better functionalities and conform to the hardware and software quality constraint Wikipedia [125].

According to Hammer [47] re-engineering is as “Re-engineering is rethinking work”. According to Yourdon [135] restructuring (for re-engineering) has
been used at least 15 years for reorganizing procedural logic of a program to the rules of structured programming. Automatic restructuring is the mechanical translation of unstructured program into a functionally equivalent structured program. Language Technology’s RECORDER was a good example of commercial product that automatically restructures COBOL programs. Other restructuring products available were SUPER-STRUCTURE, IBM’s Structuring Facility, Peat Marwick’s Retrofit. After 1985 it became fashionable to use the term “re-engineering” instead of restructuring. Whether restructuring or re-engineering the aim is to change bad system into fine system. The meaning of re-engineering is being changed, re-engineering means making the system as new development i.e. equivalent to new development. Every aspect of system is changed with modern technology, modern tools and techniques and domain knowledge. The gap between restructuring and re-engineering is widening as restructuring deals mostly with code only. Code restructuring can be fully automated and re-engineering can be done by humans using tools as assistants for routine type of work. Existing refactoring tools and techniques can cope up with complex software systems by providing abstractions and visual representations [110, 69,124].

According to Ulrich [115] re-engineering include data name rationalization, code restructuring, code splitting, code re-aggregation, and language-level upgrades. Re-engineering means up gradation of every aspect of the software system to fit in the current environment. It gives new life to the system with normal maintenance.

The first stage in Re-engineering process is Reverse engineering of the software system. Reverse engineering is the process to know the underlying technology of the software system. In reverse engineering software engineers move from code level to higher of abstractions. We can create specifications from source code by doing reverse engineering. This term also came from hardware world as it was used to know the underlying technology of the hardware systems. It is entered the software world and is working well rather more suitable for software systems. The out put of Reverse engineering process is used as the input for the next stage of the re-engineering process.

According to Chikofsky and Cross [25] Reverse engineering is the process of
analyzing a subject system to

- identify the system’s components and their interrelationship and

- create representations of the system in an other form or at a higher level of
  abstraction

Reverse engineering is nothing to do with change of the system but to
analyze the software system. Whole system is broken into individual objects/
components to understand its working completely. This operation is not done on
source code only but on design, algorithm and documentations also.

Xiaodong et al. [130] proposed a semi-automatic tool environment to
abstract the target system into higher level views more quickly which will be very
helpful for reverse engineering. As reverse engineering is part of re-engineering, it
will speed and scale up re-engineering process. The author’s work is stepwise
program abstraction and is a powerful means for reverse engineering and a
systematic approach of re-engineering, together with automatic /semi-automatic
supporting tools.

There is another legal problem in re-engineering of software systems which
must be discussed and addressed. As first stage of re-engineering is reverse
engineering and there can be legal problem in reverse engineering without the
consent of the owner of the copyright. According to Cifuentes[28] the question
arises because any form of reverse engineering involves copying of the software that
is being reverse engineered. The legal problem also depends upon the type of the
reverse engineering that is being carried out. The author defines two types of reverse
engineering as under:

- Black box reverse engineering
- White box reverse engineering

In the first type of reverse engineering, the software engineer looks into the
behavior of the software and its documentation, without seeing its internal
mechanism. White box reverse engineering involves looking at the internals of the
program to understand its working. White box reverse engineering is needed when
the former is insufficient to create a compatible program.
The development of global electronic community needed the use of reverse engineering for the interoperability of a variety of networks, software systems and re-engineering legacy software systems. The author said while discussing the legal status of reverse engineering of software that it is permitted for the purpose of interoperability, error correction and re-engineering.

Falcioni [38] tools to aid re-engineering and restructuring of legacy software are just started to appear. Although complete automation is not yet possible, the tools have substantial capabilities to assist the user in program understanding.

Byrne [19] characterized procedure for reverse engineering in a case study. It was a part of the project to develop Ada implementation of a FORTRAN source code and up gradation of the documentation. He also explained the lessons learned from experience in his study. The steps in the procedure were as under:

- Collect Information about program
- Examine information
- Extract structure
- Record functionality
- Record data flow
- Record control flow
- Review recovered design
- Generate documentation

The final step is to generate design documentation and it is an input to forward engineering process. Then forward engineering is done. Forward engineering is the process of moving from high level abstractions to code levels.

After reverse engineering, architecture of the system is transferred. New business process is accumulated and new model of business process is designed. The new architecture of software is designed according to the new business process. In this phase transformation of every aspect of software is done. BPR (Business Process Re-engineering) designs the new business process to exploit the new technology.
Jens Borchers [59] Re-engineering in all its facets is now a well recognized field within information technology literature and practice is going on for more than 25 years. Author describes his personal – and not all objective! – assessment of his experiences gained in this field in various projects and especially in the current position as head of an IT software quality department at one of the most crucial IT information service providers in Germany. According to him, the term ‘re-engineering’ and related ones like ‘reverse engineering’, ‘re-documentation’, ‘restructuring’ and ‘refactoring’, there has never been a single accepted definition. He says changes in business rules are not re-engineering.

Sneed [104] have referred to different categories as the “waves of re-engineering technology” as under:

1 The ‘Classics’:

There are two historic re-engineering initiators: the Y2K problem and the introduction of the EURO that nearly no company was able to escape.

Y2K alone created a multi-billion re-engineering market and ignited the offshore software industry in India and other countries. For many companies this was the first time they had to deal with re-engineering mechanisms.

The introduction of the EURO also created huge re-engineering requests, mainly in the European finance industry. Two significant changes had to be addressed nearly at the same time. The introduction of the EURO is to be completed by January 1, 1999 and the Y2K problem is to be resolved at latest on January 1, 2000.

2 System Platform Replacements:

The most spectacular system platform migrations took place in the late 1980s and through the 1990s,

3 Programming Language Migrations:

Language transformation is mainly caused due to lack of programmers able to maintain the used language and availability of languages on new platforms.
4 Changes of User Interfaces:

User Interfaces are re-engineered due to market competition. Screens are constructed green to win the market. Ease of use at the operational level is also increased.

5 Quality Re-engineering:

Many people want re-engineering of the system just for the quality of the system. As we have discussed above there is no universally accepted definition of software re-engineering. One author says there should not be change in functionally, the other says new functionally can be added in re-engineering process of the software system. The authors just above clearly gave the answer ‘no’ to the question, “Are re-engineering tasks allowed to modify business functionality?” Where as Van Den Brand et al. [119] says Re-engineering involves adding new functionality or developing a completely new system based on an original system’s specification using forward engineering techniques. Even the authors define four re-engineering activities as there are four maintenance activities in maintaining the software. They say re-engineering of software systems arises from corrective re-engineering needs, perfective re-engineering needs, preventative re-engineering needs, and adaptive re-engineering needs.

Authors suggest that it is important in education to impart not only a technical understanding of documenting methods, but also the ability to consider and describe new or existing systems lacking in formal documentation.

Many re-engineering approaches exist under different names. Babiker [4] grouped these approaches into three major groups. One group contains approaches that aim at improving the understanding of the software systems. A second group includes approaches that are used to improve a software system such as restructuring and data re-engineering. A third group consists of approaches that aim at migrating or developing new software systems such as language translation and extraction of reusable components. According to the author there are other approaches that facilitate activities an all groups such as design recovery and knowledge bases.
The author further suggests that re-engineering can be used to capitalize investment on current systems by extracting reusable products such as code, experience and process etc. for new development.

Business process is a set of different activities which define the pattern of work in the organizations Sethi [98]. The efficient Business processes serve better the customers and save time and efforts.

Klempa [65] Business process re-engineering innovation is a multiplicative interaction between organization’s think tank forces. The author suggests the innovation forces as organization culture, organization learning and knowledge sharing.

Kock et al. [66] gives the basic definitions and techniques for business improvement. The author stresses the importance of groupware to improve the business process. The author uses simple model(s) to emphasize his point. After redesigning the Business process, the architecture of the software is redesigned. After all type of transformations is done, forward engineering is done. The software architecture topic is to reduce the complexity through abstraction and separation of issues. Different authors have given different definitions of software architecture. Still there is no agreement on the precise definition of the term ‘software architecture’.

Eoin Woods [36] says "Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be canceled."

The third and the last stage in re-engineering process is the forward engineering. Forward engineering is moving from higher level abstractions to code levels. System is integrated in this phase and implemented. Krass [68] Forward engineering is the traditional process of moving from higher–level abstractions and logical, implementation-independent designs to the physical level implementation of a system. Forward engineering is sequence of steps going from requirements specifications to designing and implementation. Complete re-engineering process is argued by the different authors.
Rochester et al. [91] described the two re-engineering approaches as

(1) Big Bang and

(2) Trickle Down approach

In the ‘Big Bang’ approach system is shut down while re-engineering is performed. Re-engineering is performed as quickly as possible. Then the re-engineered system is installed and re-opened. This approach is no doubt quick but it is risky.

In ‘Tickle Down’ approach re-engineering is done incrementally that is group of programs are re-engineered at a time. This approach takes long time but quality checks can be done easily and timely. When clients of DST Systems Inc. needed expanded field sizes, the software system was not supporting, they had three choices as under

- Buy another new system
- Build a new system
- Reengineer the legacy system

They decided for the third choice. Then DST team was to decide for the re-engineering approach. Initially they decided for the ‘Tickle Down’ approach but due to competition they decided to go with the ‘Big Bang’ approach.

Lei Wu [70] suggests incremental model for migrating software systems. According to this model source code modules will be clustered into groups that can be migrated by a single team within reasonable time limits. Such kind of a partition of target system is therefore defined as a Migration Unit (MU), which can be dealt with as a fundamental element in the progressive migration process. The author prioritizes the MUs based on several criterions and selects one MU with the highest priority and migrate it into the target platform according to its object model. It is also being common to construct Software Product Line (SPL) by re-engineering a set of existing variant products. With this the problems of high risk and limited resources is tackled. Carnegie Mellon Software Engineering Institute defines a software product line as ‘a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment.
or mission and that are developed from a common set of core assets in a prescribed way’ Carnegie [20]. Software Product Line (SPL) is proposed as an economic way to develop and maintain the set of variant products in a specific domain Clements [30].

Gang Zhang et al. [39] demonstrate the experience and suggest that the incremental re-engineering of legacy variant products towards SPL (Software Product Line) at the component level is often a practical way for re-engineering-driven SPL adoption. However, how to properly define and carry out the increments in a steady, reduced-risk and cost-efficient manner still remains as a challenging problem to incremental SPL re-engineering. Incremental and iterative approach with stakeholder-value considerations can help to achieve steady and successful SPL re-engineering in a cost-effective manner. The authors also found that SPL adoption can be regarded as a developing result of the reconstruction and improvement of existing product assets.

Schmid[95] enlighten the reasons for the organizations to take decisions for Software Product Line. According to the authors when organizations feel that the cost of product development and maintenance grows too fast or it becomes impossible to derive new envisioned product based on the existing legacy product.

It is difficult to take decision for re-engineering, it should be based on sound economics reasons. Before re-engineering, one must know the maintenance cost, re-engineering cost, return on the investment etc. According to National Bureau of Standards US [116] following are the eleven points to make the re-engineering decision:

- When frequent system failures occur,
- When the code is over seven year old,
- When the program structure and logic flow have become overly complex,
- When the programs were written for a previous generation of hardware,
- When the programs are running in emulation mode,
- When the modules have grown excessively large,
• When excessive resources are required to run the system,
• When hard-coded parameters are subject to change,
• When it is difficult to retain the maintainers
• When the documentation has become out of date
• When the design specifications are missing, incomplete or obsolete

The following reasons are also significant for re-engineering software and are as under

(1) To bring under control the maintenance cost
(2) To increase software life span
(3) To accommodate the changed business process
(4) Migration of the system to better platform
(5) Competition
(6) Better quality service

Software system is selected for reengineering on the basis of technical quality of the system and weighs it against the business value a system still represents Jackobson [55].

Tucker [113] re-engineering is the need to improve the flexibility of existing systems and to reduce the operational costs of the legacy systems. The authors explained this in the re-engineering case study. They also mentioned the lessons learned that interim reporter who has the right experience of the software must lead the re-engineering team. Jarzabek[58] common objectives for re-engineering have been to improve program maintainability or to convert programs to a newer computer, database or language.

Oman [79] gave maturity metrics of some software system for re-engineering. According to the author following are the attributes for maturity of the software.

• Age
• Size
- Stability
- Maintenance intensity
- Reliability
- Reuse
- Subjective product appraisal

Re-engineering is rapidly being accepted among software engineers and software managers, but it is still a technology that is little understood. Re-engineering software systems can help to escape new development of software, it will bring down the graph of software investment in the organizations. From the case study of DST Systems re-engineering cost is near about one forth of the new development. Rochester et al. [92] instead of paying $50 million for a new development, which would have taken several years to complete, they reengineered the system that costs $12 million and completed swiftly.

Re-engineering is necessary to reduce the financial investments on software. Re-engineering cost is always less than redevelopment and at the same time re-engineered system is equal to the new system. Pareto law states that 80% of the problems are caused by 20% of the software (quoted by Sneed). Reengineer only that 20% of the software and leave the rest as it is.

According to Sneed, re-engineering efforts in any case are no more than 50% of redevelopment. Estimation of the cost of re-engineering software must be known to decide for re-engineering. It depends upon the size and complexity of the target system. Target system must be analyzed and broken down into parts. Sneed [103] for a typical application, these parts may be as follows on which re-engineering cost is estimated.

- Programs,
- Subroutines
- Job procedures
- Utilities
- Reports
• Files
• Data structures
• Parameters

Size of the software is measured by the number of programs, subroutines, jobs, etc or the lines of code. Size of the data is determined by the number of data objects, files, records, elementary data items. In estimating development efforts it is necessary to weight the size of the complexity. It is determined by the number of relationships between the components parts. The number of relationship should be used as an adjustment factor to take account of the number of relationship relative to the number of components.

To estimate the cost efforts of re-engineering the weighted number of units to be reengineered needs to be multiplied by the efforts required for each unit type. One should weight the efforts for each unit type by complexity factor. This effort per unit type is the re-engineering productivity which is derived from an empirical analysis of past experience.

The author derived from the experience at the Union Bank of Switzerland, in which 2000 statements program required 0.5 person days for re-engineering using the tool SOFT-REORG. At the Union Bank of Switzerland near about 50% of the cost mounted up after the programs were converted and transferred to new environment.

The formula from the author’s point of view can be derived as under:

Weight the efforts of unit/module by complexity factor of the unit n (say), let it be W_n,

Re-engineering efforts for the unit n from the past experience let it be E_n

Then sum up total re-engineering efforts are as under:

E = \sum W_n*E_n \quad '*'\quad is\quad multiplicative\quad sign

Cost of re-engineering can further be reduced by using the automated tools. IT (Information Technology) works as a good deriver and lifter for re-engineering of software systems. Software engineers can design automated software tools for
various stages of software re-engineering process. This cost calculation for re-engineering was for structured/procedural programming environment. The re-engineering cost of object oriented software systems will be further lesser as the reusability of the objects/units /modules is more.

The author of the thesis proposed a model to estimate the cost for re-engineering of object oriented software systems. The estimated the re-engineering cost as 25% of the cost of new development. The research paper entitled ‘Cost of Re-engineering (Object-Oriented Software Systems) versus Developing new One- A Comparison’ was published in the journal of ‘International Journal of Software Engineering and Computing’ Vol. 3 No.1, January-June 2011.

Re-engineering has numerous benefits over new development. Sneed [103] calculated the benefits of re-engineering over new development as under:


There are two aspects, one is value (performance) increased and the other is decreased expenditure incurred by reengineering.

In the above equation the benefits even can be negative even. This will happen when there is much difference between the new value and the old value of the system. In this case reengineering should not be preferred as it will not be economical. In another way it can be simple as under:

System benefit value (say SB) = New Value of system – Old Value of system


Both calculated variables SB and CB can not be negative.

The net benefit in reengineering is = SB + CB

SB (System Benefit) means increased performance of the system and decreased maintenance cost.

Re-engineering is done to reduce complexity and the maintenance cost. Maintenance starts after the delivery of the software product. Maintenance cost goes on increasing with the age of the software system. According to Erlikh [37] more
than 90% of software life cycle costs are spent on maintenance. Some time uncontrolled and unmanaged expenses on software worsen the condition. Lientz et al. [71] also found that many organizations were spending 20%-70% of their computing efforts on maintenance. In such situations, re-engineering can potentially save money for the organizations. The maintenance activities are of four types. Bennett [8] the mean distribution of maintenance efforts in 487 DP (Data Processing) organizations was as under:

- Perfective maintenance 50%
- Adaptive 25%
- Corrective 21%
- Preventive 4%

Slovin [100] maintenance cost is reduced by 50% after re-engineering. This is quoted by the author (Slovin and Malik) in the case study of re-engineering of large insurance company’s ageing COBOL based system. The result was given as under

- Corrective maintenance reduced to 50%
- Number of hours spent on emergency maintenance reduced by 50%
- Number of trouble reports reduced by 60%
- Batch processing time has been reduced by half (50%)

In literature, the field of estimating software maintenance cost is not widely covered. We could be dependent upon valuable contributions [102, 101, 72, 136, and 78].

Buchmann et al. [15] presented cost estimation model which was implemented on Deutsche Post MAIL. Deutsche Post MAIL is Europe’s largest postal provider, processing mail and parcel in Germany as well as world wide. According to authors software maintenance was 26% of the total information technology budget in 2009. The authors provided a maintenance cost estimation model as under
Cost estimation = f (factors influencing maintenance cost) where f is an algebraic function to combine and weight individual metrics describing meaningful factors.

Jingyue et al. [61] gave 12 the most influential effort/cost drivers for software maintenance. A few are worth to mention here as size of the maintenance task, kind of changes, maintainer’s confidence, maintainer’s experience etc.

Software maintenance workload is very large and is the problem faced by developers, engineers, software managers and users. Ren et al. [87] says that many software development organizations in foreign countries use 60% of the manpower for the maintenance of existing systems. The increase in the number of software systems is further increasing the workload of maintenance.

Software re-engineering is a complex and difficult process. Re-engineering legacy software system requires a highly trained staff with experience in the current and the target system, automated tools and the specific programming languages.

Ruhl and Gunn determined that the complexity of the re-engineering process increases in relation to the complexity of the programs. The complex programs need more manual efforts. In order to investigate the feasibility and cost-effectiveness of re-engineering legacy code the author performed the case study. The application system was provided by the Internal Revenue Service (IRS). NIST (National Institute of Standards and Technology) USA conducted a competitive procurement to award a Labor Hours contract for performance of the case study. The ceiling price for the contract was set at $250,000. The application system was near about 50,000 lines of COBOL programming.

The findings Ruhl [93] was as under

<table>
<thead>
<tr>
<th>Program Group</th>
<th>Automated</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>96%</td>
<td>4%</td>
</tr>
<tr>
<td>Batch with SORT</td>
<td>90%</td>
<td>10%</td>
</tr>
<tr>
<td>Batch with DBMS access</td>
<td>88%</td>
<td>12%</td>
</tr>
<tr>
<td>Batch with SORT and DBMS access</td>
<td>82%</td>
<td>18%</td>
</tr>
<tr>
<td>Interactive</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>
Level of efforts for each step in the methodology was as under

<table>
<thead>
<tr>
<th>Steps</th>
<th>Percentage of efforts</th>
<th>Total Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline Current System</td>
<td>19.76%</td>
<td>780</td>
</tr>
<tr>
<td>Analyze</td>
<td>43.26%</td>
<td>1708</td>
</tr>
<tr>
<td>Produce Documentation</td>
<td>4.05%</td>
<td>160</td>
</tr>
<tr>
<td>Generate new code</td>
<td>26.34%</td>
<td>1040</td>
</tr>
<tr>
<td>Execute and test</td>
<td>6.59%</td>
<td>260</td>
</tr>
</tbody>
</table>

The author finally recommended that ‘It is critical that the application system experts be involved throughout the re-engineering process. They are essential for design recovery.’

Software re-engineering can be cost-effective and viable solution for extending the lifetime of an application system. The degree to which it is cost effective is dependent on the goals for re-engineering, the condition of the legacy application system and documentation, available automated tool support, and the involved personnel.

Sneed [105] in restructuring program the maintenance cost reduced only 5% where as in re-engineering the application software maintenance cost reduced to 17%. The author found in his case study that re-engineering can decrease complexity and increase maintainability where as restructuring has only a minor effect on the maintainability. The author found that size of the program is reduced by 31% through restructuring and 32% through re-engineering.

Hanna [49] the problems with legacy systems are many and varied. The difficulties revolve around three of the major reasons for trying to alter existing code:

- Correction of existing errors
- Accommodation of new business requirements, and
- Upgrading to take advantage of a new technology
Dave Sharon, president of Case Associates, Inc., Oregon, Ore., said the problems are multiplied by the number of people touching the code.

Charles Bachman, founder and Chairman of Bachman Information Systems, Inc., Burlington said that the re-engineering is not any thing which ever finishes and describe six steps that constitute the software re-engineering cycle:

- Capture
- Reverse Engineering
- Enhancements
- Forward Engineering
- Optimization
- Generation

Bachman believes that although the entire process is not fully automated, ‘a good number of individual pieces of the puzzle have solutions’.

Van De Vanter [118] emphasizes the importance of comments and white spaces for comprehensibility and maintainability of source code. Program comprehension is vital for reverse engineering and reverse engineering is first phase in re-engineering.

Pretty-print is stylistic formatting convention to source code which is very useful for source code analysis. These formatting rules usually consist of changes in positioning, spacing, color, size and similar modifications intended to make the content of source code easier for people to view, read, and understand. Pretty printers for source code are sometimes called code beautifiers or syntax highlighters.

Pretty-Printing in the software re-engineering area serves two purposes. Firstly, for automatic software re-engineering, it is used for source (re-)generation, to transform the abstract representation of a reengineereed program back to human readable textual form Merijn [73]. Programs are first parsed, then reengineered and finally pretty-printed. Secondly, for semi-automatic software re-engineering pretty-printing is used for documentation generation Van Deursen [120]. In this case, the re-engineering process requires user intervention and documentation generation is
used to make programs easily accessible.

To support source generation and documentation generation, a pretty printer should be able to produce multiple output formats. Morcos Chounet E. and Conchon A. define formatings independently of such output formats in pretty print rules.

Program understanding is necessary in both maintenance and re-engineering of software systems. All the tools and techniques that are required to understand program are double benefitted one for maintenance and the other for re-engineering. Cleveland [31] presents a tool PUNS (Program Understanding Support environment) for understanding a program. Program understanding is needed for software maintenance. The author says that software maintenance represents the largest cost element in the life cycle of software. Cleveland says 50 to 90 percent of the total life cycle expenditure is done on the maintenance. 50 percent of the maintenance cost (efforts) goes to program understanding. The tool discussed by Cleveland supports the program understanding task by organizing and presenting the program from many different view points. The tool detects the low-level relationships that exist within the program using static analysis techniques. It consolidates and organizes these relationships and presents them in a user-friendly environment. Re-engineering software project’s success depends heavily on how well the software is organized, because the organization affects understandability, modifiability and testability. Software changes rapidly during maintenance and its organization deteriorates.

Corbi [32] program understanding can be done in three actions: read about the program (read documentation), read it (read the code), and run it (observe the execution). While execution, recurring interaction patterns occur at various levels of abstractions. Visualizing these patterns in program execution can facilitate program behavioral understanding which will be helpful in design recovery for re-engineering tasks Jerding[60]. Programmers are needed thoroughly understand the foreign code for software maintenance and re-engineering.

In Object-oriented systems, a class is a collection of related data types and operations. It is called a module. Cohesion is the degree of interdependence of
components of a module. Coupling is the degree of interdependence between modules. According to Baowen[6] a good software design is that which has high cohesion and low coupling. The author says that the low cohesion makes the system difficult to understand. High cohesion objects can be extracted easily and therefore can be reengineered individually.

Weiser [123] was the first to introduce the idea of program slicing which is very helpful to understand the foreign code. Program slicing is a technique for restricting the behavior of a program to some specified subset of interest. A static program slice $S(v,n)$, of a program $P$ on variable \textquote{v}, or set of variable identifiers at statement \textquote{n} yields the portions of the program that contributed to the value of \textquote{v} just before the statement \textquote{n} is executed. $S(v,n)$ is called a slicing criteria. Slices can be computed automatically on source programs by analyzing data flow and control flow. This slicing technique can be used to form decomposition for software systems.

In contrast, Korel [67] defined a dynamic program slice that is computed with respect to a specific input in \textquote{v} and is only valid for that input. A dynamic slicing criterion was defined to consist of a set of variables, a line number in the program code and a sequence of input values. Static slicing techniques can also be used for analogous purposes, but the computed static slices would be relatively old-fashioned. This poses a serious limitation on the applicability of such static model slices.

According to Wuyts [129] gaining understanding of a large scale industrial program is often a daunting task. For this dynamic analysis has proven it\'s usefulness for gaining insight in object-oriented software. But even then, collecting and analyzing the event trace of large scale industrial applications remain a difficult task.

The Authors proposed an intermediate representation for software architecture by extracting relevant information from UML structural and behavioral model(s) into a single system model. This model represents various types of dependencies existing among different model elements. They named this representation the Model Dependency Graph (MDG). The authors proposed
algorithm Dynamic Slicing of UML Architectural Model(s) (DSUAM). It is based on traversing the edges in MDG for any given slicing criteria.

According to Demeyer et al. [35] change the broken parts of the software system only. Broken parts of software are those ones which puts the software system at the risk. The authors also suggest for changing most valuable part first. They said ‘Most Valuable First’ will help you determine what priority to give to problems in the system, and will tell you which problems are on your critical path.

A number of CASE tools that aid in program understanding have been designed and proposed in the literature. The tools PBS, Rigi, CIAO, DECODE extract design information from software system and store it in a knowledge base. According to Wong, ‘Rigi’ is good system for analyzing software systems through the reverse engineering technique developed at the University of Victoria Wong [127]. As size of software system grows large it become more difficult to maintain. To improve maintainability and reusability of a software system the identification of coarse-grain or fine-grain components and the consequent modularization of the system becomes a promising solution for migration tasks. In the re-engineering and software migration domain, the methods that are mostly considered for identifying major system components include clustering, slicing and objectifying Hammer [47].

Miguel [74] CodeCrawler is a language independent reverse engineering power full tool which combines graphs and metrics to generate views of object-oriented systems. It supports reverse engineering of large object-oriented systems. It enriches a simple graph like tree with metric information of the object-oriented entities it represents. All the characteristics of CodeCrawler were mentioned by the author in his Ph.D. thesis.

Schwanke [96] describes an intelligent tool for re-engineering software modularity. Re-engineering modularity includes both discovering the latent structure of existing code and changing that structure to obtain better modularity. This tool provides heuristic modularization advice for improving existing code. A heuristic design similarity measure supports two services:

1. Clustering
2. Maverick Analysis
Clustering identifies groups of related procedures and Maverick Analysis identifies individual procedures that appear to be in the wrong module.

Boehm [11] 80% of programming resources are allocated to maintaining and re-engineering existing code, not to developing new applications. Therefore there is need for re-engineering and maintenance tools to analyze and understand the large body of existing code and then modify it so that it accommodate new requirements. Burson et al [16] presented an approach that is machine capturing, analysis and transformation of the software system. This approach significantly extends the opportunity for automating maintenance and re-engineering. The approach uses REFINE toolset and is centered on the following points:

1. Object-oriented database to capture existing software.
2. Data definition and query languages that support program analysis.
3. Syntactic pattern-matching against program templates expressed largely in the target language.
4. Program transformation rules that automatically modify existing code or generate new code.

The program transformation system ‘REFINE toolset’ supported automation in re-engineering software systems.

Britcher [14] describes his re-engineering experience of software system TRACON (New York terminal approach control application software that supports air traffic control in the New York City and other areas). The contract was given by FAA (Federal Aviation Administration) to three companies for $ 2.5 millions. It was successfully completed in nine months. The following factors which identified as key contributors to the success of the project were reported to FAA:

- Software tools and laboratory was available, tools were easy to learn and use.
- The team followed standard software development life cycle and used consistent and proven set of methods.
- Software architecture was recorded.
- A complete data dictionary was prepared to map the variables to
reengineered system

- It was reengineered in Pascal/VS from Ada

The following tools were used:

- ISPF/PDF – an interactive editor and library manager
- REXX – an interpretive language
- SQL/DS - relational database management system, an automated document preparation package

In this study the author found that re-engineering software system over reinventing is significant.

Re-engineering tools presently are needed for centralized environments. Existing tools for this type of environment are inflexible and non-scalable. Chiang [23] with the advent and wide spread use of object-oriented and client-server technologies, customers are expecting their software to take advantages of these new technologies. Software Vendors have two approaches to renovate their software for the new technologies as under:

- Convert
- Facelift

The first approach is to re-engineering the legacy software as a new one and the second approach is to facelift their products by means of middleware Clemens[29] such as CORBA Randy [86], or COM/DCOM. Chia- Chu Chiang emphasizes on the facelift approach by distributed object computing. It is environment in which objects interact by passing messages to each other. Object Request Broker (ORB) works as the glue that ties the distributed objects together. The ORB makes available the techniques for the objects to locate and activate other objects on a network.

As the complexity of software systems and the demands on quality, safety and maintainability has grown considerably over the last decades. Vangheluwe [121] has proposed Model-Driven Engineering (MDE) which treats model(s) in various formalisms as first-class artifacts. Such model(s) may be obtained by reverse-
engineering of existing software artifacts for the purpose of analysis, optimization and evolution. Fundamental modeling language concepts are elaborated in Giese [40].

Chung et al. [27] presented case study of Bertie3 as web services shows that modernizing a software system with web services will permit the business components of the system to be easily expanded and integrated with other application components for future demand. The reengineered version of Bertie3 is SoBertie3 (Service –Oriented Bertie3).

Data re-engineering is very important for re-engineering software systems. Data problems can be divided into two classes:

1. Data definition problems
2. Data values problems.

1 Data definition problems are as under

- Inconsistent names
- Inconsistent field lengths
- Nonflexible field lengths
- Inconsistent record layout
- Inconsistent data dictionary etc.

2 Data value problems are as under

- Changes in data validation rules
- Problems of defaults values
- Negative values becomes positive if sign omitted
- Data type mismatch problems etc.

Ricketts et al. [89] recommends that methods and tools for data definition re-engineering problems are available in the market and that they must be used to solve variety of data definition problems. The author tells that some data definition re-engineering tools can change source code. According to the author re-engineering data definition can increase the maintainability of exiting software. Data definition
re-engineering is a precursor to data re-engineering. And finally the author advised that data definition re-engineering along with procedure re-engineering and data value re-engineering prepares an organization for reverse engineering. Ignacio et al. [54] proposed a tool specifically designed for database re-engineering. The tool produces a diagram representing the possible conceptual schema used during the initial development of the database. An ER or EER diagram is useful when the final software product will be a new version of the database.

Hevner [52] proposed a method for data re-engineering in structured programs. The strength of the method is based on the theories of box structures and regular expressions. The structured program is viewed in a referentially transparent hierarchy of modules, decomposed via control structures. The usage of each data item is defined in regular expressions and mapped onto the hierarchy. Based on these abstractions, the program data flow is reengineered to eliminate data use anomalies, reduce data scope, and form data objects. The significant advantages brought about by data re-engineering include increased program understandability, improved maintenance, and evolution characteristics and in many programs, better performance.

Database re-engineering is also a part of the software re-engineering. Database re-engineering should be parallel with the other re-engineering activities. Strobl et al. [108] talked about the mid sized in-house Information System (IS) that was 30 years old system in which COBOL VSAM files migrated to RDBMS.

Hainaunt [45] historically identifies three periods in DBRE (Data Base Reverse Engineering). The first period, eighties, was devoted to migrating CODASYL databases, IMS databases and standard files to relational database technology. The second period was to refine elicitation techniques to recover implicit constructs and to develop more flexible methodologies to address the problem in all its complexity. In the present decade, the increasing consensus on XML as data model, the view of the web as an infinite database, the use of dynamic SQL in most web information system, the increasing use of ORM(Object-Relational Mapping) environments etc. that make data reverse engineering more necessary and more complex by an order of magnitude.

42
Analyzing the structure of large programs is one of the most frustrating and time consuming parts of software maintenance and re-engineering. Chen et al. [21] describes a program abstraction system designed to provide different levels of abstractions of software systems that can aid the discovery process during software maintenance. The authors built the C Information Abstraction System (CIAS) to analyze many production programs at the University of California, Berkeley.

CIAS could be used to study the following aspects of program structures:

- **Subsystem**: Identify self-contained components in a large software system.
- **Layering**: Topologically sort different combinations of reference relationships.
- **Dead Code**: Detect unused code in a software system.
- **Coupling**: Analyze the binding strength between pairs of software objects.

CIAS system consists of three major components:

- The C abstractor
- The Information viewer (InfoView)
- And software investigator.

The authors mention several other program abstraction systems that have been proposed and reported in the literature as under:

- **MasterScope**
- **FAST**
- **OMEGA**
- **Cscope**
- **CLF (The Common Lisp Framework programming Environment)**

Users of CIAS have found the system helpful in discovering various aspects of their programs. However the authors also learned and mentioned few lessons from building CIAS. Re-engineering projects still have considerable risks and difficulties. Much of the technology required in the strategic re-engineering lifecycle is
available, but no tool environments exist to support strategic re-engineering in an integrated way. Re-engineering process must be masterminded and conducted by humans, with tools acting as assistants capable of performing routine work. Advances in research on automated programming understanding may allow us in the future to better support re-engineering activities that today relay on human intelligence [10, 51, 50, 88, and 117].

A major re-engineering objective is to reuse software. Reusing software assets may help organizations go forward with less risk and less cost than creating new systems from scratch. The basic problem in reuse software is how to acquire reusable assets of old software. Software assets are software parts, software designs and software specifications. Software parts are like functions, procedures, and data type and value definitions. Software designs are program flowcharts, procedure flowcharts, structure charts, and data flow diagrams etc. Specifications are like business model(s) and rules. Other assets are like test data, code templates and documentation about software.

Arnold [2] describes software salvaging as a re-engineering specialty for recovering software assets for reuse. Software salvaging helps populate a repository with parts and relationships, or help recover object-oriented objects and classes from non-object-oriented software. The author talked of transforming a part to make it more reusable. According to author reusable parts once obtained must be certified for quality. The author gave metrics for measuring reuse effectiveness for C system. To estimate these measures, the following information is needed.

- The abstraction hierarchy
- A definition of external repositories
- Definition of the “uses” relationship, (means what part use what other parts)

According to the author following information is needed for each part being used:

- Name of the part
- Source of the part (internal or external)
- Level of abstraction
Usage (means the number of times an item is used)

The authors suggest that reuse levels are much higher than is generally assumed. The author discovered that total average reuse for the C systems were 58%. Bailey and Basili [5] have proposed software–cycle model for reuse and re-engineering. The model suggests five stages to reuse a component:

• Analysis of existing programs to sort components to be reused.
• Reengineer to eliminate domain specific troubles.
• Save reusable components in the repository.
• Construct independent status components with a reusing approach and store in a repository.
• Reuse components to develop new programs.

Reuse of software components is not free of cost, it is actually very hard to reuse any piece of code unless a fair bit of effort was put into designing it so that it could be reused [108]. It is also essential that investment in reuse requires management commitment to put the right organizational infrastructure in place, and should only be undertaken with clear, measurable goals in mind Goldberg [42].

According to Haddad [44] software organizations have to invest huge sums of money to start successful reuse methodology and it is a barrier for them. In his opinion, the core of reuse is source code. The author estimated that domain specific components represent up to 65% of the application size and one approach to effective reuse practices focuses on domain specific components.

Re-engineering models proposed by software engineers and researchers were very useful for re-engineering. Horse Shoe Model was proposed by Bergey et al. [9], Carnegie Mellon Software Engineering Institute CMU/SEI, TN, 2000. CORUM-II was contributed by Woods et al. [128]. It was very helpful in integration and interoperability of re-engineering tools. E. J. Byrne [18] proposed a model which was very flexible for changes at various levels of reengineering. Yang et al. [133] present a dual spiral model making the re-engineering process more effective and efficient. The work flow in this model requires two systems to work together. One is the
legacy (candidate) software and the other target system (reengineered software). Go on moving the functionality from the legacy system to the target system step by step as in the spiral model of software development life cycle. In the whole process, the active functionality in the legacy system is in a decrementing pattern, and the active functionality in the new target system is in an incremental pattern. It facilitates adding new features while transitioning the existing functionality. A proxy system in needed to facilitates the whole process. The advantage of this model is that the step by step spiral procedure adds new functionality into the target system. This model adopts cyclic approach and is less risky.

Rainfall Model for re-engineering only the faulty portion was proposed by Singh et al. [99]. Reflexion model which enables a software engineer or developer to summarize a large body of source code in terms of a selected, high-level, structural view of the system was given by Murphy [77]. Alessandro et al. [1] proposed ‘iterative re-engineering method’. This model suggests re-engineering work to be iteratively executed on various components in different phases. During the process there will be coexistence of legacy components, components currently undergoing re-engineering, reengineered components, and new components, added to system to accommodate new functional requirements. This model improves the re-engineering procedure in a good way still it has some deficiencies. The quick change of business environment for legacy systems requires a shorter re-engineering cycle. Frequent development of code for transition systems will cause resources wastage and increase the complexity. And lastly cost on format change between old and new data module in the iterative operation will impair the performance of transition system.

Xing Su et al. [131] proposed parallel iterative Re-engineering Model for Legacy software systems that is faster than the “iterative re-engineering method” proposed by Alessandro et al. The former model uses rapid re-engineering process and uses full development resources. It allows several components and shares data to be migrated in one cycle according to their access relationships. But with the change in software development technology, existing models were less applicable and need change. These models, their applications and short comings were studied in depth.
Stephane [107] proposed a pattern form as the appropriate style communicating expertise about the re-engineering software systems. They say re-engineering patterns are stable units of expertise which can be consulted in any re-engineering effort dealing with object-oriented software systems. Patterns document re-engineering processes which will be useful in re-engineering object-oriented software systems.

Chiang [22] proposed a software stability model using re-engineering. Software stability reduces the impact of software changes by dividing a system into stable modules and unstable modules. Changes made to the unstable modules for software evolution should not impact the stable modules. Software stability model minimizes the maintenance efforts due to changes.

Hamza [48] presents an approach for designing a system that is less likely to change over time using the software stability model.

Bush [17] presents an approach assessing requirements stability using scenarios. To conduct requirements stability analysis, a goal list and a set of scenarios need to be developed first. The scenarios are then analyzed for stability against the goal list.

Tonu et al. [112] proposed metric-based approach to evaluate software architecture for software stability. According to the authors, the architectural stability can be evaluated using growth rate, change rate, cohesion and coupling of the system.

Park et al. [82] proposed object-oriented model refinement technique for software re-engineering. According to the authors, domain knowledge is represented as an object model in re-engineering for object-oriented re-architecture. In this technique one object model is generated from domain knowledge and the other object model is generated from reverse engineering. Effective comparison and refinement of the two object models is done.

Chung et al. [26] stressed on Service-oriented Software Re-engineering (SoSR) methodology. SoSR is achieved by applying Service-Oriented Computing (SOC) to the legacy software systems. According to the authors this methodology is
very helpful in conducting re-engineering of tightly coupled, rigid, and non-service-oriented legacy software systems into loosely coupled and service-oriented software systems. Compared to other middleware technologies such as RMI, CORBA and mobile agents, SOC can design software systems that are more easily integrated to the legacy systems as they have standardized service interfaces and interaction protocols. Weerawarana [122] Service-Oriented Computing (SOC) has emerged as a software development paradigm that enables the design of software system as a set of services, called Software as a Service (SaaS). Rs 40 lakh software is running on computer system of Rs. 40 thousand is common practice. The most of the IT budget now goes in supporting the software assets. The support cost includes Annual Maintenance Cost (AMC), keeping the software secure and virus free etc. SaaS is cost effective and faster software services accessed via a web browser. It makes a case for pay for usage of software rather than owning software for use [41,114].

Beck et al. [7] proposed a visual analysis and design tool for planning software re-engineering. This tool eases estimating efforts and change impact of a planned re-engineering. The authors conducted three case studies on industrial software systems to demonstrate usage and scalability of his proposed approach.