CHAPTER - V
PROPOSED RE-ENGINEERING MODEL(S)

5.1 Introduction

Software re-engineering process modeling is the activities of representing the process of re-engineering software systems. There exist software re-engineering models designed by software engineers and researchers from time to time. Most of the existing re-engineering models work in stand alone way, and were not working at a common platform. The object-oriented paradigm has been adopted in many software industries and in the coming future object-oriented software would exist everywhere. Therefore it was the need of the hour to design the model(s) for re-engineering object-oriented software systems.

The characteristics and working of the existing models were studied. The applications, usefulness and limitations were reviewed. Models do not exist for all re-engineering processes to complete re-engineering life cycle. There exist many gaps in the re-engineering process life cycle. Re-engineering process life cycle is complete re-engineering from identifying the software for re-engineering to implementation of re-engineered (renovated) software. Another problem with existing models is that most of the re-engineering models were for re-engineering programs in procedural languages. As software development technology is changing fast these were less applicable for object-oriented software systems.

After studying thoroughly existing/available models in the literature, efforts were done to design new model(s) for re-engineering object-oriented software systems. The model(s) are systematic and uniformly cover the whole re-engineering process life cycle. These are contributed to both communities the software managers and the software engineers. Proposed re-engineering model(s) will be very useful to enhance the re-engineering process for object-oriented software systems. The proposed re-engineering models are given below:

1. Models for differentiating ‘maintenance’ and ‘reengineering’

2. Decision Making Models for Re-engineering
3 Efforts Comparison Model
4 Cost Estimation Model
5 Rainbow Model for Re-engineering
6 Model for Increasing Software Age

The above cited models are explained as below

5.2 Models for differentiating ‘maintenance’ and ‘reengineering’

From literature review, it was seen that there is confusing over re-engineering and maintenance terms in small and medium sized software development and client companies. This confusion is one of the problems in encouraging software re-engineering in the companies of small and medium levels. Software managers are often confused over ‘maintenance’ and ‘re-engineering’ and they experience little difference in these two terms. Maintenance and re-engineering terms are closely coupled with each other and some time used interchangeably. Where as re-engineering is very much different from maintenance and the difference is increasing as re-engineering is potential and is taking importance in the coming future. The idea of software re-engineering came latter than maintenance in software industries. Re-engineering is for making the system new according to the state-of-the-art technology, environment and new business process where as maintenance is to keep the system operational. This confusion is hurdle for software managers in deciding for software re-engineering. These two terms are yet young and developing. One is the problem in the development of the other in respect to the subject matters. It is time to differentiate the two terms and promote the subject matters on these two concepts.

These two terms came from the world of hardware objects. Now theses entered the world of software and are well suitable for software systems. Future scope of these two areas will be vast as software will exist everywhere. And at the same time changes in the requirements, business process and technology are inevitable.

It is difficult to draw a clear cut line between these two terms. Re-engineering of software systems is a topic of importance and in coming time it will
be gaining attention in the world of software industry. These two terms should be separated to promote the subject matter because one is problem for the other. It is tried here to put them in different non overlapping regions. Maintenance and re-engineering are two different areas in software engineering. Maintenance is for making the software operational till the age of the system where as the re-engineering is to change the software to make it new to have a fresh life span. Scope of re-engineering is vast and challenging as compared to maintenance. Re-engineering is to reduce the expenses on software systems in the organizations. Re-engineering has more scope in the world of software than in the world of hardware’s as software do not depreciate. Maintenance is close to repairing/mending where as re-engineering is very close to new development.

These two terms are yet young and developing and gaining importance. One is the problem in the development of the other in respect to the subject matters. Now software is gaining importance in every sphere of life and these two are very closely associated to software system life cycle. It is time to differentiate the two terms and promote the subject matters of these two concepts.

In this work these issues will be studied and it will be very useful for the software industries. It will help software managers to recognize and make the best use of these terminologies for treatment of their software systems.

5.2.1 Model based on Software Age

The age concept for software systems is different from other type of systems. Today’s software can be old tomorrow. It is old when our business process changes. On the other hand, as software never depreciates, it can work for long time. According to literature review, the age of software programs in procedural languages is well thought-out is 7 years and age for object-oriented software systems is 10 year as object-oriented software systems are more maintainable. Software system will work with normal maintenance cost till it gets old. Re-engineering is not feasible when the software is not old.

One measure of the age of software is the number of faulty objects in the software system. The objects which do not conform to our changed or new business process requirements and other requirements such as efficiency, cost, and ease of
use, life and security and do not give the desired output are called faulty objects otherwise fine objects. Further, faulty object hangs without responding or if responding to some operation, its response may be incorrect. It is a threat to the working of software system.

Software should be re-engineered only when it is old/age. Software is old when it met any number of the following points.

- When the code is 10 years old
- When the maintenance cost is high
- When it becomes overly complex
- When system failure is common
- When the number of faulty objects is greater than half
- When business process is changed
- When there is a big gap in hardware and software technology

When software system does not meet any of the above situation, it is not old and it is maintainable with normal maintenance cost. Every system needs maintenance for the whole life period. Maintenance starts in the beginning (at the time of delivery) and goes on increasing with the age of the software. When maintenance is not possible it means software life is ended. There are two alternatives now either retire the software or re-engineer the software. Re-engineering reconstitute the software to a new one and it can work for another span of life.

5.2.2 Model based on Software Life Cycle (SLC)

Maintenance and re-engineering are closely related and glued to each other. When maintenance exhausts, re-engineering of software is done. Maintenance is the last stage of the software development life cycle (SDLC). Software goes through various stages till it retires. Software Development Life Cycle (SDLC) defined in the literature already exists. In this period software is in developing state. It is the period from the inception of the problem to delivery of the software.
Software Life Cycle (SLC) is defined as a period from software delivery to retirement of the software. In this period software is always in the operational/working state. It consists of two stages software maintenance and software re-engineering. After delivery, software remains in the loop of these two stages till it retires as shown in the figure 4.1.

Software maintenance, the last stage in software development life cycle (SDLC) should be extracted from the cycle and put in the SLC (Software Life Cycle) as first stage. Re-engineering should be added as the second stage in Software Life Cycle (SLC). In this way, SLC will be consisting of two stages the first maintenance stage and the second re-engineering stage. Maintenance stage starts just after the delivery of the software. Software remains in the maintenance stage till it gets old. When software gets old, re-engineering stage starts. Re-engineering cost goes on increasing after maintenance stage. When software is re-engineered, software enters in the maintenance stage again and this loop continues till software is retired. Software will be in the loop of the two phases of maintenance and re-engineering after development.

Software Life Cycle (SLC) is proposed in this work to differentiate the maintenance and re-engineering. In this way, there will be two cycles in the history of software. After completing the first cycle (SDLC) software will be in the second
cycle (SLC). The software is in the first cycle when it is in developing state and the second cycle is when the software is in working state.

5.2.3 **Discrete Model for Differentiation**

This model differentiates the two concepts on the basis of discrete points which clear the jurisdiction of both the concepts. Following are the tables which show the differentiation based on the various issues.

**Cost and need based factors**

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Software Issue</th>
<th>Maintenance</th>
<th>Re-engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>When software is delivered</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2.</td>
<td>When software is not old</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3.</td>
<td>When software is old</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>4.</td>
<td>Number of faulty objects less than half</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>5.</td>
<td>Number of faulty objects greater than half</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6.</td>
<td>Cost comparison (software is not old)</td>
<td>Feasible (Less)</td>
<td>Not feasible</td>
</tr>
<tr>
<td>7.</td>
<td>Cost comparison (software is old)</td>
<td>More (high)</td>
<td>feasible</td>
</tr>
<tr>
<td>8.</td>
<td>Cost comparison (Transitional state)</td>
<td>High</td>
<td>Feasible</td>
</tr>
<tr>
<td>9.</td>
<td>Cost level</td>
<td>Comparatively less</td>
<td>Comparatively high</td>
</tr>
</tbody>
</table>

Table 5.1
### Architectural and Functionality Factors

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Software Issue</th>
<th>Maintenance</th>
<th>Re-engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Reverse engineering</td>
<td>Needed</td>
<td>Needed</td>
</tr>
<tr>
<td>2.</td>
<td>Architecture</td>
<td>No change</td>
<td>May or may not Change</td>
</tr>
<tr>
<td>3.</td>
<td>Forward Engineering</td>
<td>Not needed</td>
<td>Needed</td>
</tr>
<tr>
<td>4.</td>
<td>Business process</td>
<td>No change</td>
<td>May or may not Change</td>
</tr>
<tr>
<td>5.</td>
<td>Addition of attributes</td>
<td>May or may not</td>
<td>May or may not</td>
</tr>
<tr>
<td>6.</td>
<td>Additions of objects</td>
<td>May or may not</td>
<td>May or may not</td>
</tr>
<tr>
<td>7.</td>
<td>System change</td>
<td>Renovated/repaird</td>
<td>Like new development</td>
</tr>
<tr>
<td>8.</td>
<td>Type of activity</td>
<td>Patch up</td>
<td>Like new development</td>
</tr>
</tbody>
</table>

### Other Factors

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Software Issue</th>
<th>Maintenance</th>
<th>Re-engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Software age</td>
<td>Life span increases</td>
<td>New life span begins</td>
</tr>
<tr>
<td>2.</td>
<td>Subject Area</td>
<td>Software engineering</td>
<td>Software engineering</td>
</tr>
<tr>
<td>3.</td>
<td>Man power</td>
<td>Skilled</td>
<td>Highly skilled</td>
</tr>
<tr>
<td>4.</td>
<td>Volume of work</td>
<td>Comparatively less</td>
<td>Comparatively more</td>
</tr>
<tr>
<td>5.</td>
<td>Risk</td>
<td>less risk</td>
<td>High risk</td>
</tr>
<tr>
<td>6.</td>
<td>Time</td>
<td>Comparatively less</td>
<td>Comparatively more</td>
</tr>
<tr>
<td>7.</td>
<td>Software managers (Interest)</td>
<td>More</td>
<td>Less</td>
</tr>
<tr>
<td>8.</td>
<td>Software engineers</td>
<td>More</td>
<td>less</td>
</tr>
<tr>
<td>9.</td>
<td>Origin</td>
<td>Hardware world</td>
<td>Hardware world</td>
</tr>
<tr>
<td>10.</td>
<td>Future scope</td>
<td>Equal</td>
<td>Equal</td>
</tr>
</tbody>
</table>

Table 5.2

Table 5.3
Re-engineering of software systems is not common practice as yet. Many people are confusing over these two terms and take theses two terms as similar terms in the small and medium sized software development companies and client companies. Many software people use them interchangeably. This confusing is a problem in the way of development of these two areas. Maintenance of software is more common than re-engineering, but re-engineering has more potential over maintenance.

These models are proposed to expose the hidden potential of re-engineering. Re-engineering can be two ways benefit to the society. On one hand it lowers the financial expenditure on software and on the other it saves the technical efforts and reduces the software backlog. The software managers are often confused and could not make the right decision at the right time. The applications of these models are to know whether the software is to maintain or re-engineer or retire. Required treatment of the software can be done according to these models.

**Important Features**

Three models are presented to differentiate the potential terms re-engineering and maintenance as separate non overlapping regions in software engineering. It will help in promotion of subject matter of both these areas.

### 5.3 Decision Making Model(s) for Re-engineering

The ability to accurately estimate time and efforts for re-engineering software is the key factor for successful of re-engineering project. And more important is the right time decision for re-engineering software system. Right time decision affects not only the cost of re-engineering but also possibility for re-engineering. If we cross this crucial point, re-engineering cost will be more and some time can not be possible. We can come across at a stage when there is no option other than purchasing new software. But when is the right time? What is that crucial point? It is that stage when software is best fit for re-engineering. As we go past this stage, re-engineering will be costly. Accumulated affects of maintenance makes the system complex and deteriorate the system’s architecture. When maintenance cost is too much high or difficult to maintain, it means system is to retire or re-engineer. Re-engineering software reconstitute it into new form to work
for another life span. Re-engineering should be done at right time. If we overlook this time, re-engineering will be costly or even some time not possible and then we have to throw the costly legacy software and purchase new software.

How the right time is to be determined? These issues are addressed in this model. This is very useful for Software managers and they will be aware of this point after going through this piece of work. Right time for re-engineering is a point in software life cycle and is called the decisive point.

Significance of Decisive Point

‘Decisive point’ is the time in the software life cycle at which the software system is best fit for re-engineering and re-engineering cost in minimum. Software Life Cycle (SLC) is the period from software delivery to retirement of the software. Software managers must focus on this point for re-engineering the software to increase the life span of software. Maintenance plays an important role in the life cycle of a software system. After the product has been released, the maintenance phase keeps the software up to date with environment changes and user requirements changes. After a certain period, a stage comes when it is difficult to maintain the system. Then re-engineering is the solution. Re-engineering is the only way to avoid new development cost. Re-engineering and maintenance are confusing terms and overlapping each other (section 5.2). In this type of situation, right time is ignored and organizations have to go for new development. This problem of software managers is addressed in the previous models. If software managers do not know the right time, how can they go for re-engineering?

Model(s) for Decisive Point

Three model(s) are proposed which are very useful for the software managers in the organizations (client companies)

5.3.1 Thoroughfare Decisive Point

In this model, life span of software is divided into two zones as in the following figure 5.2

- Maintenance zone.
- Re-engineering zone.
Software product is delivered at the point A, and Maintenance of the software starts at the beginning (when the software is delivered) at the point A as in figure 5.2 and is minimum cost at that point.

Software system is candidate for maintenance in the first zone from the point A to D and candidate for re-engineering in the second zone from point D to point B. We can not maintain the software in the re-engineering zone and at the same time re-engineering is not feasible in the maintenance zone due to unjustified costs or other reasons. After the point B, re-engineering can not be done and software managers have to retire the software.

![Diagram](image)

**Fig. 5.2 Re-engineering/Maintenance zone**

As in figure 5.2 above, red point D in the middle of the life span is vital point, best fit time for re-engineering. The point D is transitional point where maintenance zone ends and re-engineering zone starts. Point D is reached when software age is 10 years old and/or failure rate is increased. Software Managers must reengineer the system at the point D to have another life span of the software. If software managers do not reengineer the system and maintenance zone goes overlapping the re-engineering zone with high maintenance cost, they will struck at a situation when maintenance is not possible and re-engineering cost is also not feasible (high re-engineering cost) and there is no alternative except to retire the software system. According to the above figure 5.2 the software can not work beyond the point B and the software managers have to retire the software and new software is to be purchased. Maintenance cost is maximum at the point D and re-engineering cost will be optimum (minimum cost) at the red point D. After the point D, re-engineering cost is increasing. Therefore software managers must stick to this point D for re-engineering and to escape new costly software development.
5.3.2 Decisive Point based on Maintenance cost

In this model, decision for re-engineering is done on the basis of maintenance cost. Following figure 5.3 depicts the graph of maintenance cost of the software system during the life cycle of software. Maintenance cost starts from the point O (Origin) in the beginning when software is delivered. It goes on increasing with the age of the software. Software age in normal situation is ten year long. After ten year software is aged (old) software. In the following figure, software is aged at the point D and maintenance cost starts increasing sharply. At this point software should be re-engineered to bring the maintenance cost down or retire the software. After the point D, maintenance cost increases with high rate and therefore it is difficult to maintain the software with such a high cost. If we retire the software then we have to bear the cost of new software. Cost of new software is much higher than the cost of re-engineering (section 5.4 and 5.5).

![Maintenance cost curve](image)

**Fig. 5.3** Maintenance cost curve

In the above figure, Maintenance cost is normal from point O to Point D and is slightly increasing. Software is not old from point O to D. After point D maintenance cost is increasing sharply because software is aged. From point D onward software is aged and needs re-engineering. Beyond the point D in the figure, it is difficult to maintain the system with such a high maintenance cost. At this point system should be reengineered or retired. If we reengineer the software at this point,
Re-engineering cost will be optimal. The point D in the figure is significant for the software managers. Reengineered Software will be new one with another life span and Maintenance cost will be ordinary (small) for another life span of the software.

5.3.3 Decisive Point based on Faulty Objects

This is object based model for decision making about re-engineering of software. It is to be determined on the basis of the faulty objects (section 5.2.1). Re-engineering identifies reusable components (objects) and analyzes the changes that would be needed to regenerate them for reuse within new software architecture. The use of a repeatable, clearly defined and well understood software objects, has make re-engineering more effective and reduced the cost of re-engineering. Decisive Point will also be determined on the basis of faulty objects.

Let our candidate system be an object oriented system with N objects. Architecture of the software system deteriorated with age and maintenance, objects start becoming faulty with every patch up to the software system. These faulty objects can be reengineered and can be plugged again to get the system reengineered. Let the N objects in the software be O1, O2, O3,..................On. As the system ages the faulty objects goes on increasing. Go on maintaining the software till half of the objects are not faulty. When half of the objects (N/2) are faulty in your application go for re-engineering the software. The re-engineering cost of the candidate system with near about N/2 faulty objects will be optimal.

Decisive point (right time for re-engineering) is when exactly N/2 objects become faulty

When N/2 objects become faulty, Re-engineering Zone starts. This is vital time in the software life cycle for re-engineering. If the software managers pay no heed to this time, re-engineering cost go on increasing with the increasing number of faulty objects. A time comes when re-engineering zone finishes and there is no alternative except to retire the legacy software.

Software managers should not ignore Decisive Point otherwise they have to retire the under utilized software system.

These model(s) are valuable to software community for re-engineering the
software systems at the right time. Re-engineering is not feasible before and after the decisive point.

**Important Features**

On the basis of the above presented models, software Managers can know the right time for re-engineering. If re-engineering is done at the right time, the cost of re-engineering will be optimal (minimum) cost.

These models will be working as an alarming bell for re-engineering software systems at the right time

**5.4 Efforts Comparison Model**

In this model total efforts done on re-engineering some software and total efforts done on developing software of equal domain were compared. The efforts of re-engineering of object-oriented software systems are often significantly less than the efforts of developing new software of equal domain. Object oriented software systems are more reusable and therefore it will also help in reducing efforts. Re-engineering of software systems rather than developing new software will save precious time and resources. Re-engineering reduces the cost of maintenance by increasing the software quality and reducing complexity. To justify re-engineering, the efforts of re-engineering software must be estimated and compared with the efforts of new software (new development) of equal domain. The efforts of re-engineering depends upon many factors but major factors are the part of the software (number of objects) to be reengineered and complexity (interrelationship between objects) of the software. Efforts comparison model is presented and on the basis of this model re-engineering can be seen as efforts saving in the software industry.

The cost of maintenance increases as the software gets evolved. Complexity of the software also increases. In early computing days software maintenance was twice as expensive as developing the software. It was that time when computer machines had become many folds powerful and software was inefficient. This was referred to as legacy software crisis by Seacord et al. [97]. There is strong relationship between software quality and maintenance cost. For every dollar spent on quality in development, a dollar is saved in maintenance. Cost of re-engineering
also depends upon the software quality. Now software developers are meticulous about software quality. Object-oriented paradigm has changed the scene for re-engineering. Object-oriented software system is all about objects. Object is the basic component of any system. Here in this context, it is abstraction of module of the software that can be independently designed, tested and plugged into the rest of the software. It is a bundle of data and instructions interfaced with the software system. The idea of object is useful for software engineers to design quality software with ease. Object can be big or small it depends upon the number of attributes in the object and number of attributes in the object depends upon the requirements of the problem. The concept of object is equally important in designing as well as in re-engineering the software system.

5.4.1 Comparison of Efforts is based on objects

The idea of object oriented software systems came from object oriented physical systems (natural and man made systems). The Mighty Nature has created object oriented systems as better service and maintenance is possible for object oriented systems. Now software engineers designed object oriented software systems on the pattern of physical systems. The life of object oriented software systems can be increased by re-engineering the software system. Physical systems can not be reengineered again and again as after certain time the base architecture and the objects will be wear and tear out or weaken. Because physical objects depreciates with time. But this does not happen in the case of software systems. Software system never dies nor depreciates, but when the business process, technology and/or environment change we have to re-engineer the software or retire the software.

Limitation of human capacity for dealing with complexity is the need to break up the system to understand and deal with it. Object oriented system can be understood easily as each object embodies an amount of complexity that programmers can fully grasp. In successive generation of languages decomposition is supported differently. The advancement in the software technology, from machine language to assembly language, assembly language to procedural language and from procedural language to object-oriented languages, helps programmers to make their
jobs easier accordingly. Object oriented technology uses objects and not algorithms as its fundamental building blocks. In this context, object is a conceptual module that can be plugged in and plugged out from the software system.

Objects in the object-oriented software are like real world objects. Objects have boundary and communication lines to send/receive messages. The object in the system is characterized with three properties: Identity, State and Behavior. Identity distinguishes it from others, state is the data stored in it and behavior describes the methods by which the objects can be used.

Objects are visible as separable, independent packed in a system. Object-Oriented technology is big jump in the software engineering field. Objects help in managing increasing complexity and change in the software by replacing and adding objects/modules. In object-oriented systems, the idea of abstraction is of much important for mastering complexity. The topology of object-oriented languages helps us in building blocks which represent the logical collection of classes and objects instead of subprograms. It helps in understanding existing software system for maintenance and re-engineering as well as to develop new software. In particular, object technology is very useful for re-engineering object oriented legacy software systems. Data and operations are united in such a way that building blocks of our systems are no longer algorithmic but classes and objects. In object model emphasis is positioned on characterizing the components of the system modeled by a programmed system.

5.4.2 Metrics for Object-Oriented software systems

Object-oriented metrics plays key role in measurement of software system which will help further in measuring parameters relating to re-engineering for example re-engineering cost etc. OO (Object-Oriented) metrics are applied to analyze the source code. Metrics are very useful in evaluating the quality and complexity of the software products and the software development process. Large number of methods per object increases the size and complexity of the object.

To estimate the cost of re-engineering the size and complexity of the software must be known. The target system must be broken into its components parts. Object oriented design metrics play vital role in designing quality software.
Object-oriented software system, at higher level of abstraction, is viewed as collaboration of objects. To find the cost of re-engineering object oriented metrics play very important role. It will help in identifying the objects their complexities and interrelationship between them. We need to recognize all the aspects of objects to be reengineered.

In an object oriented context, Chidamber et al. [24] proposed a set of following metrics

- **WMC**: Weighted Methods per Class
- **DIT**: Depth of Inheritance
- **NOC**: Number of Children
- **CBO**: Coupling Between Object Classes
- **RFC**: Response For a Class
- **LCOM**: Lack of Cohesion in Methods

Different groups of authors suggested different design metrics, thorough study reveals that no one set is good in all respects. But with good metrics it is possible to identify the Objects, their interaction, cohesion and coupling and hence to understand the software system. Metrics are used to analyze, articulate, judge and predict the characteristics of the software. With the help of good quality metrics re-engineering cost, re-engineering time, re-engineering risks etc. can be estimated. The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.

### 5.4.3 Efforts Comparison Process

In this model, efforts of re-engineering and efforts of new development of equal domain are compared. It is clear from the literature available that efforts of re-engineering of software system are often considerably less than the efforts of developing new software. It is applicable to all the systems (may be physical system, software system or any other type of system). The above statement is applicable if re-engineering is done at the right time. There is a critical point beyond which re-engineering cost is high and is some time even difficult and risky. Because
accumulated effects of maintenance worsen architecture of the system.
William M. Ulrich justifies the cost of re-engineering and argues that about 25% of the cost of new software system is needed for re-engineering the legacy software system. He quoted the example of DST systems case study (DST Systems Inc. a software development firm). Instead of paying $50 million for a new system, which would have taken several years to complete, they accurately analyzed their existing system, and decided to make changes on it that cost $12 million, and were completed rapidly [115].

5.4.4 Re-engineering efforts based on faulty objects

Re-engineering efforts depend upon the number of objects to be reengineered. There is direct relationship between the efforts required and the number of faulty objects in the software system. In proposed models, re-engineering is based upon individual objects and only faulty objects are re-engineered for re-engineering the whole system. If the number of faulty objects increases, the required efforts also increase. The relationship between objects and efforts required is depicted by the following figure 5.4. In the worst case the efforts of re-engineering software can be equal to the efforts of new development of equal domain.

![Diagram showing relationship between efforts and number of objects]

Fig. 5.4 Relationship (Efforts and faulty objects)
Software system is not always candidate for re-engineering; it can be maintained till it completes its life span. When software is old, it is candidate for re-engineering. Software is old, when it is of 10 year and/or our business process changed. The plug-in and plug-out of the object (software module) is possible as in the world of physical systems (section 5.6.3 object modeling). For re-engineering the software system, it is to be identified the objects to be Added/Deleted/ Modified/ re-engineered. On average half of the total objects would be reengineered if we are under the critical point that is we are re-engineering the system at the right time. Object oriented systems are good candidates for re-engineering due to reusable feature of object oriented systems. The plug-in and plug-out of the objects is possible, and faulty objects in the system are extracted (plugged out), reengineered and fitted (plugged in) again. 50 % of the objects in the system can be faulty which are to be re-engineered for re-engineering the software system.

An object is re-engineered at the cost of 50% because of the reusability of the code in the object. It means 50% of the 50% of total efforts of the new development software development are needed for re-engineering the entire software system. Therefore in normal situation on average 25 % of the efforts of new software system (new development of equal domain) are needed for re-engineering legacy Object-Oriented Software System. The following things are assumed for the above value of re-engineering efforts

1. Documentation of software is available.
2. No major change in the business process
3. Software system is not more than a decade
4. Original developer(s) would make changes (Re-engineering) and they have the source code available.
5. Re-engineering is done at the right time
Figure 5.5 Shows efforts for re-engineering and efforts for new development software of equal domain.

From the above discussion, it is clear that re-engineering of existing system is usually cheaper than to replace the software system with new development. 75% of the efforts for new system are saved according to this model. In the above model it is tried to estimate the difference between efforts of re-engineering and efforts of new software based on the number of objects to be reengineered (faulty objects). The efforts difference is the direct financial benefit to the organizations in re-engineering the software systems instead of buying new systems.

Object oriented software development gave significant shift in system design because objects tend to model artifacts that exist in the real world. The objects can simulate the exact behavior of real world objects. With this practice, time and efforts would be reduced significantly. The quality and reliability of the software produced with object-oriented practice is drastically high. Object oriented software systems are good candidates for re-engineering and less efforts are involved.
This is a conceptual model based on the behavior of the real world objects. According to this model the efforts of re-engineering a system are one forth of the efforts of new system approximately with the conditions mentioned above with the model.

**Re-engineering efforts = 1/4 of Efforts of the new development**

Future work can be done to verify this model empirically. Primary data can be collected from the software developers and /or from the client industries which orders for re-engineering object oriented software systems or orders for new software system to verify this efforts comparison model.

This work will be very helpful to software managers and engineers to know the difference between the two jobs re-engineering and new development. They can easily decide for re-engineering or new development. Comparison is based on faulty objects. It will be a justification and clear cut agreement between the two communities (software companies and client companies) for re-engineering software systems. The difference in efforts is of great concern for the organizations as it is a direct reduction on the software investments. The right time of re-engineering is that when the faulty objects (section5.2.1) in the system are at the most 50% (section 5.3).

**Important Features**

This model compares the efforts in re-engineering and developing the software of equal domain. As there is a lot of difference in efforts in doing these two activities, it will expose the over all benefits of re-engineering.

**5.5 Cost Estimation Model**

The estimation of cost of re-engineering of software is a complex process. There is no universally accepted model or method for estimating re-engineering cost. On the other hand, estimation of re-engineering cost is the primary factor to promote re-engineering. Re-engineering cost must be known objectively to the software managers to come forward for re-engineering. No doubt re-engineering cost will be certainly less than the cost of developing new software. Re-engineering of software systems rather than developing new software will save precious time and
resources. Re-engineering reduces the cost of maintenance by increasing the software quality and reducing complexity. To justify re-engineering, the cost of re-engineering software must be estimated for reengineering projects and it should be less than the cost of new development. The cost of re-engineering depends upon many factors but major factors are the portion of the software (number of objects) to be reengineered and complexity (interrelationship between objects) of the software. In this work efforts are done to present a re-engineering cost estimation model. On the basis of this model, software managers can negotiate with the software engineers to take a decision whether for reengineer or to retire the software.

The ability to accurately estimate the time and cost of re-engineering software is the key factor for successful of re-engineering project. Cost estimation is needed before re-engineering is initiated. The primary objective is to enable the client and software engineer to perform a cost benefit analysis. The estimate can be in terms of person-month (PM), which can be converted into cost by multiplying the person-month rate. Many factors in estimation are not quantifiable. Also re-engineering area is young and needs much maturity and improvement. Quality of software design matters in the process of estimation. Easiness in software understanding, maintenance and re-engineering depends upon the decomposition of system.

Object-oriented technology is more helpful in measuring re-engineering cost as it uses objects which are handled independently. Re-engineering identifies reusable components (objects) and analyzes the changes that would be needed to regenerate them for reuse within new software architecture. The use of a repeatable, clearly defined and well understood software objects, has made re-engineering more effective and reduced the cost of re-engineering. Good objects are easy to handle as if some object is pulled out from the software system, software system will be working without much affecting the whole system except the job done by that particular object. As in the other physical systems, a component is plugged out, repaired and plugged in the system again. Additional screws, nuts and bolts are required for this purpose; same must be developed as a universal language to fit an object in the system.

In this work, it is assumed that Object can be seen at a higher level of
abstraction and is taken as independent module or unit that can be plugged in or plugged out of the software system without much affecting the software (section 5.6.3 object modeling). Software is delivered at the site and maintained, with time and maintenance effects software complexity increases. Large systems are complex having more objects as each additional object increases the complexity of the system Halladay [46]. Objects become faulty with changes in requirements. The number of faulty objects increases with software age. As the number of faulty objects increases, re-engineering cost increases. Candidate software system is disbanded; all objects of the system are separated. Faulty objects are identified and collected for re-engineering. Old design of the software is examined (Reverse Engineering). Then redesigning of the structure of the system to improve the quality of software system is done (transformation of the architecture). According to new quality design, objects are integrated (Forward Engineering).

5.5.1 Right time for Re-engineering

Right time for re-engineering is necessary to minimize (optimal) the re-engineering cost. Software managers can easily know the right time for re-engineering of the legacy software system (section 5.3). Re-engineering cost depends upon the number of faulty objects in the candidate software. Re-engineering cost increases as the number of faulty objects increases. Objects become faulty with increasing age of software and with effects of maintenance. With recurring maintenance, complexity increases and software quality decreases. As the software is maintained, errors are introduced. Many studies have shown that each time an attempt is made to decrease the failure rate of a system, the failure rate got worse. On average, more than one error is introduced for every repaired error. After a certain period, there is a crucial point when it is difficult to maintain the system or maintenance cost is too much high. Then re-engineering of the software is done. Maintenance problems are a driving force behind re-engineering. The cost estimation model will help the software managers to compare the quoted cost of new software with cost of re-engineering. Certainly re-engineering will attract the software managers because of big difference in the cost of re-engineering and new software development.
5.5.2 Process for estimating re-engineering cost

Re-engineering of object-oriented software systems is done by re-engineering objects individually and independently. Object-Oriented software system is being more reusable and hence more suitable for re-engineering. Reusability will affect the cost and time of re-engineering. To justify re-engineering, the cost of re-engineering software must be estimated and it should be less than the cost of purchasing new software with a great difference. Only then the re-engineering alternative will be considered feasible. The cost of re-engineering depends upon many factors but major factors are the defective portion of the software to be reengineered. Complexity (interrelationship between objects), size and business process change also effect the cost. Cost model will help organizations in making the decision for re-engineering. This is a major decision faced by the software managers. It is not simple to calculate the re-engineering cost as well as new development cost. Efforts are done to present a cost estimation model for re-engineering legacy object oriented software system.

To estimate the cost of re-engineering the following factors are taken into consideration.

- Number of objects to be reengineered.

- Size of the object (Number of attributes in the object)

Each object has its own attributes, but attributes are taken into consideration according to requirement specifications and business processes. For example attributes of object employee are ‘name’, ‘employee identification code’, ‘address’, ‘mobile phone’, ‘landline phone’, ‘age’, ‘height’, ‘color’, ‘basic pay’, ‘grade pay’ and many more. If software is required for payroll of the employee, the attributes like ‘height’, ‘age’, ‘color’ etc are not required. Number of attributes depends upon the size of the job done by object.

As re-engineering cost depends upon the number of objects to be reengineered, it means re-engineering cost of an object is to be estimated to estimate the re-engineering cost for the entire software system. The candidate object is the faulty object. On average half of the objects could be candidate objects (section 5.3).
Following is the figure 5.6 of software system with some (seven) faulty objects.

![Diagram of Application System with faulty objects](image)

**Fig. 5.6 Application System with faulty objects**

Application software system (in figure 5.6) can be maintained by modifying the faulty objects and need not re-engineering. Number of faulty objects increases with the age of software. Re-engineering cost heavily depends on the number of faulty objects. These faulty objects can be reengineered and can be plugged in to enhance the functionality of the software.

### 5.5.3 Re-engineering Cost of the Candidate Object

Traditional software measurement techniques are not satisfactory for measuring productivity and predicting efforts for developing software systems. The Source Lines of Code (SLOC) metrics and the function point metrics both were for programming environment putting the data and procedures separate. In object oriented paradigm data and procedures are combined. In object oriented approach the role of UML i.e. Unified Modeling Language is supreme. It was designed to provide a standard for software modeling languages. It is a graphical notation for object-oriented analysis and design. UML provides a framework for describing a set of model(s) that capture the functional and structural semantics of any complex information system. UML constructs in object oriented software can be used for estimation of resources like efforts & cost etc. While calculating the efforts of re-engineering it is important to include information about communication between objects and reuse through inheritance to the size of the object (Lines of code) as well. An object is small piece of source code that can be maintained or reengineered separately.
Common and simple approach for measuring efforts for developing small software with single variable is as given below:

\[ \text{Efforts} = a^* \text{(SIZE)}^b \]

where a and b are constants to be determined by regression analysis applied to historical data Jalote [57]. SIZE is a variable and the value of this variable depends upon the size of the software. The SIZE is the number of lines of code. This model is for the structured (procedural oriented languages) software systems. This model measures the efforts for developing new software systems.

It is desired to measure the efforts for developing an object. The size in case of an object is the number of attributes required for developing an object. Efforts for re-engineering small piece of source code as an object can be calibrated as under

\[ \text{Efforts} = \left[ a^* \text{(number of attributes)}^b \right] / 2 \]

Where a, b are constants and can be determined from historical data (from past re-engineering experience) of reengineered software systems. If we denote number of attributes of an object involved in computations by A, then the above model will be as under

\[ E_1 \text{(Efforts)} = \left[ a^* (A)^b \right] / 2 \text{ Person Months for an object O1} \]

This model estimates the total efforts for re-engineering an object, a small piece of code that can be plugged in with the object oriented software. Cost of re-engineering all the faulty objects will be estimated by adding all above such E's. Let us suppose there are n faulty objects then cost of re-engineering of all the objects will be \[ E_1 + E_2 + E_3 + \ldots \ldots E_n. \]

**5.5.4 Re-engineering Cost of Candidate System**

After estimating the cost of all the individual faulty objects, re-engineering cost of the software system will be estimated. The system is reverse engineered (section 5.2.1) objects are identified and separated (section 5.6.1) for re-engineering. Cost model for re-engineering an object is presented above (in section 5.5.3) as efforts for re-engineering an object \( O_1 \) will be \( E_1 \). With this model, re-engineering cost of all the faulty objects is estimated separately. Let us suppose our candidate system is with n faulty objects say \( O_1, O_2, O_3, \ldots \ldots \ldots O_n \). Re-engineering cost
of all the n objects will be added and is equal to $E_1 + E_2 + E_3 + \ldots + E_n$. Now all the objects are fine and we need to integrate them into a system. At this stage software architecture will also be redesigned (improved). There will be additional efforts (cost) for identifying the faulty objects, transformation of the architecture and integrating them into the new design. We denote it by $C_n$, the constant is to be determined after verifying the results by empirical data available from the past reengineered projects. Then the total efforts for re-engineering the candidate software system will be as under

$$E = E_1 + E_2 + E_3 + \ldots + E_n + C_n$$

$E$ is total Person Months of re-engineering of software system with $n$ faulty objects. Efforts $E_1, E_2, E_3, \ldots, E_n$ are person months of re-engineering $n$ faulty objects $O_1, O_2, O_3, \ldots, O_n$ respectively. $C_n$ is constant which will be determined by applying this model to the past reengineered software systems to fit in the model. $C_n$ is additional cost for redesigning the architecture and integrating all the objects into the modified design.

Person Months estimated can be multiplied by the current rupee rate of Person Month work.

Estimated re-engineering cost = Estimated Effort $\times$ current Rate of Person-Month

If we denote current rate in rupees for person months by $R$ then cost model will be as under

$$\text{Cost} = E \times R$$

In this piece of work cost model for re-engineering object oriented software system is presented which will be valuable to organizations regarding software investment.

This model is indispensable to organizations as they can settle the deal for re-engineering software and can escape buying the new software.

The future work is to test and apply this model to real situations and to find the suitable values for the constants used in this model. The constants ‘$a$’ ‘$b$’ and ‘$C_n$’ used in the model are unknown and to be determined on the basis of past data of re-engineered software systems. Empirical data from reengineered object-oriented
software systems and the actual cost incurred can be seen, verified and suitable values could be assigned to the constants. 40-50 projects are sufficient that can be examined to fit the values of the constants in the model. With suitable values of above constants in discussion, the re-engineering cost model for object oriented software systems is ready. This model will facilitate re-engineering process for object oriented software systems.

Important Features

The cost estimation model is of great importance for the organizations for re-engineering their software systems. They can estimate the re-engineering cost and can decide for re-engineering. If you do not know the cost of some product, it is difficult to prepare the mind for acquiring that product. When you know the cost of re-engineering, it will be very easy for the organizations to plan for re-engineering software.

Beneficial to both the communities, software managers (client companies) and software engineers (software development companies) to negotiate for re-engineering object oriented software systems.

5.6 Rainbow Model for Re-engineering

Rainbow Model for re-engineering object oriented software systems is for re-engineering the faulty objects only instead of re-engineering the whole software system. Faulty objects are re-engineered individually. In this way, it is economical technique for re-engineering software systems.

The objects which do not conform to our changed or new requirements and other requirements such as efficiency, cost, and ease of use, life and security and do not give the desired out put are called faulty objects and otherwise fine objects. Further, faulty object hangs without responding or if responding to some operation, its response may be incorrect. It is a threat to the working of software system.

This Model is contributed to the field of re-engineering which will be useful for the organizations to reduce in software investments. This model is also useful to software engineers as it saves the efforts by re-engineering faulty objects only instead of re-engineering the whole software as in the case of the existing models.
The software companies are currently facing the tremendous problem of attempting to maintain a huge legacy of existing software while developing new software systems, all of which must be accomplished in the face of shrinking budgets and fewer skilled personnel. You can’t afford to throw the legacy software but to maintain or reengineer the software. If maintenance cost is high then think of re-engineering. Reengineered software will work for another life span with normal maintenance cost. If there is no concept of re-engineering then this costly legacy software will be wasted and new software will be purchased which will be financial burden for organizations. On the other hand legacy/old software will be wasted underutilized and added to the software backlog. In this way software backlog will be increasing and organizations will have to make heavy investments on software.

In this technique, software is reverse engineered and faulty objects are identified for re-engineering. Re-engineering has three main stages namely reverse engineering, architecture transformation and forward engineering. The same three stages are applied for re-engineering every faulty object individually. Object is reverse engineered to recover the design of the legacy object, then modification or change of the design is done if needed and it is forward engineered using new technology, new software language, in the new environment.

5.6.1 Process for Identification of faulty objects

The process of identifying the faulty objects is given below in fig. 5.7

![Diagram](image-url)
As each software system is composed of the separate modules, we can separate out the modules and test to identify the faulty modules for re-engineering. Software system is fed to the modules separator to identify and separate all the modules for individual testing. All the modules identified by the module separator are fed to the Sieve as shown in the above fig.5.7.

Sieve tests each module individually to separate out the faulty modules and fine modules. When separation is done, faulty objects are collected for re-engineering and fine objects are collected to use as it is in the new software. Threshold technique can be used in Sieve. Each module is tested for all the factors like efficiency, speed, security, etc based on the dynamically adjusted threshold. Dynamically adjusted threshold can be made to adjust automatically with the changing software requirement and according to the module being tested. If any module fails to meet the threshold requirements it is separated out as the faulty module and requires re-engineering.

5.6.2 Re-engineering Model for an Object

In the following figure 5.8, model for re-engineering a software object is given on the basis of which faulty objects are re-engineered individually.

![Diagram](image)

**Fig. 5.8 Re-engineering Model for an object**
In the above model the faulty object is reverse engineered and design specifications are recovered. Output from reverse engineering process is input to the forward engineering. Then forward engineering is applied taking this input (design specifications). Forward engineering is the third and last stage in the re-engineering process and is to rebuild the object. That re-engineered object will be like new object called fine object.

5.6.3 Object Modeling

In object modeling, models are presented to plug-in and plug-out the objects in the software systems after re-engineering individually. In object oriented software system, objects are created, verified and are packed. Objects are tightly coupled or loosely coupled. Nature of the coupling depends upon the relationship and interface among the objects. Object can be big or small it depends upon the number of attributes of the object used in the programming. The concept of object is equally important in designing as well as in re-engineering the software system.

Degree of coupling of objects also matters. Higher degree of coupling cause difficulties in software re-engineering as it increases complexity. Independent objects are more reusable. The following figure 5.9 shows a software system with three objects.

![3-objects system](image)

**Fig. 5.9** 3-objects system

Object oriented language should be such that objects can be easily extracted and plugged in the software system. Software object should be like a hardware object to fit in the rest of the system. What makes the objects good to plug-in and plug-out with more ease is as under:

1. Well documented
2. Complete
3 Independent (Low Coupling)
4 Cohesive
5 Certified
6 Having well defined interfaces
7 Conforming to a component model
8 Useful
9 Secure
10 well defined boundaries
11 Fine object
12 Tested

The above features make the objects flexible and useful for re-engineering.

In the following figure 5.10 there is no addition in functionally of the software but there is change in the functionally. No new object is added in the system. One object is re-engineered and again plugged in the software system.

![Diagram](image)

**Fig. 5.10** Changed functionality

In the following figure 5.11 there is no change in original functionally but addition of functionally is done. New object is added in the system.
5.6.4 Re-engineering Process with Rainbow Model

Software is decomposed and architecture is recovered. Faulty objects (section 5.5.1) are identified and separated. Faulty objects are collected for re-engineering and fine objects are collected to use as it is in the modified design. Faulty objects are re-engineered individually (section 5.6.2). The re-engineering process for re-engineering the candidate software system has been shown in Fig. 5.12. In re-engineering software system we move from one end point (legacy system) of the rainbow to the other end point (new software system) to get reengineered software system.

Rainbow is an arc in the sky of spectral colors, usually identified as red, orange, yellow, green, blue, indigo, and violet, that appears in the sky opposite to the sun as a result of the refractive dispersion of sunlight in drops of rain or mist. Components of lights are bundled into a one system called rainbow. In this model rainbow is considered as a set of many standard programming languages supporting and compatible each other. This is called re-engineering environment for re-engineering identified faulty objects. Faulty objects are sent to this environment to make them fine.

Faulty objects are reengineered using modern languages and techniques which are available in the rainbow re-engineering environment. Reengineered faulty objects become fine objects going through re-engineering environment of the rainbow and are received at the (other end) right end of the rainbow model. Fine objects identified at the time of reverse engineering are collected directly to the right end of the rainbow. Now all the objects are fine and are collected at the right end of the rainbow model. Now it is time of architecture transformation. Architecture of
the software is redesigned according to the new requirements, changed business process and/or new environment and technology.

Objects are integrated into new design. Objects are plugged-in and plugged-out into the new architecture according to the work done in section 5.6.3 (Object Modeling).

Fig. 5.12 Rainbow Model
Rainbow Model for Re-engineering
The object-oriented system which is candidate system for re-engineering is viewed as a system of faulty and fine objects. Software is re-engineered when faulty objects are just more than fine objects. In this technique software system is reverse engineered as we move from left end upward. In reverse engineering, software architecture and all the artifacts of software are extracted. In the above figure 5.12, a software system with five objects is our candidate system. Software system is untied and objects are identified to be re-engineered according to changed requirements. The other objects are taken as it is to plug in the modified design. Here in the above model, legacy software system consists of 5 (five) objects. Three objects named 2, 3, and 4 are faulty and 1, 5 are fine objects. Faulty objects are sent to re-engineering environment as depicted in the figure 5.12. The faulty objects 2, 3, 4 become fine after re-engineering and are received at the other end of the rainbow. The objects 1 and 5 are taken as it is to the other end of the Rainbow. Then all independent objects are integrated to make a system – new system. In the above model software system having the five objects is reengineered.

**Important features**

Rainbow model is for re-engineering for object oriented software systems. Rainbow model saves re-engineering efforts as it aims to re-engineering of only selected (faulty objects) components/ Objects. One forth efforts are required for re-engineering (section 5.3). It will be more suitable in future as we are going towards changed coding style and languages with higher and higher level abstractions and hence designing of objects will be better-quality with the help of IT (Information Technology) derivers. The objects would have good interface to plug-in and plug-out in the software systems (section 5.5.3).

Rainbow model is economical model as it does not re-engineering the whole software, it picks the candidate (faulty objects) objects only for re-engineering.

Most of the existing re-engineering model(s) and methods are applicable to structured monolithic software systems, as the object technology came later. Back log of structured software systems is increasing and creating problems like wastage on software investments. Object-oriented programming is becoming the dominant
programming methodology, re-engineering model(s) and techniques are required for re-engineering object oriented software systems.

Rainbow Model is best suitable for re-engineering object oriented software systems. This model works on object based re-engineering. In this model the candidate (system to be reengineered) system is reverse engineered and all the source code is interpreted in the form of individual objects at the higher level of abstractions. Then objects are identified to be reengineered (changed). The identified objects are re-engineered individually. Individual objects are re-engineered on the pattern of procedural software system with the re-engineering model for an object (section 5.5.2). The main features of the model are as under

- Re-engineering is based on individual objects
- No need to re-engineering the whole software system
- Additional functionality can be added by simply adding objects
- On average half of the software needs re-engineering (section 5.3).

It is re-engineering cost saving model, as software engineers can use it to re-engineering the defective part of the software only. This model is most suitable for re-engineering object oriented software systems which will be helpful to software engineers.

5.7 MODEL FOR INCREASING SOFTWARE AGE

Software age can be increased manifold by re-engineering again and again when it gets aged. Software starts aging after it is delivered for operational use in the field. Evolution of software over years makes the system complex. Re-engineering decreases the complexity and brings down the maintenance cost. Reengineered system is new one with the values of all the parameters slightly less than the values of the parameters of new development. Every time when maintenance cost increases beyond the certain level, re-engineering of the system can be done for bringing down the maintenance cost for another life span. It will help software managers to utilize fully the legacy software system for long time.

Software system is changed for better performance and long life. There is a
set procedure to change the system called software maintenance and re-engineering. Software maintenance and re-engineering are different for hardware systems (physical systems) because software doesn't physically wear out, but often gets less useful with age. Maintenance is done to keep the software working in proper order and re-engineering is done to bring down the maintenance cost.

Re-engineering reduces the complexity of software and improves the business process to fit in the new environment and technology. Re-engineering of software systems is a sub area of software engineering area which is young and growing. Software engineers must develop new tools, and methodologies to cope with the challenges of maintaining large and aging systems. Our aim here is that if there are some changes in the requirements the program system should be maintained or reengineered. If there is no concept of re-engineering then there will be software backlog everywhere. This will be wastage of resources and time in the software industry.

Research in the field of software re-engineering can ease this burden by recovering the design of these legacy systems and re-implementing the legacy systems in a format that allows for easier maintenance and support.

Age concept is different for software systems, it can work for years as it never depreciate or tear out. But on the other hand today’s software system may be old tomorrow, it depends upon the requirements and required out put. When our requirements change or business process change or due to any reason it does not meet our requirements or required output then it is called aged. When software is delivered, its normal age (life span) is well thought 7 years for structured oriented software and 10 years for object-oriented software system. No need of re-engineering till the software is aged. When software is aged, re-engineer the software at right time to use it for another life span. Re-engineering cost will be minimum cost if the software is re-engineered at the right time. When software is old/aged? When is the right time for re-engineering? These are the problems of software managers. The solutions to these problems are discussed in the previous sections (section 5.2).
Software managers can re-engineer the software system at the right time with help of the models given above in the section 5.2. The re-engineering cost will be the minimum cost at that time.

Below is a model for increasing software age manifolds by re-engineering the software system at the right time when it gets old. Re-engineered software can work for another life span with normal maintenance.

![Diagram showing software life spans](image)

**Fig. 5.13, showing software in fourth life span**

In the above figure 5.13, ‘M. Cost’ means maintenance cost and O is the origin. This figure is showing the maintenance cost of the software in successive life spans. Software is delivered at the point O and it gets old at the point R1. From point O to R1 is the life span of software. Point R1 is taken as the right time for re-engineering (section 5.2) and software is re-engineered at the point R1. The maintenance cost comes down as shown in the above figure, and the software is ready to work for another life span with normal maintenance cost. After 10 year again software is aged at the point R2. Taking it as the right time for re-engineering software is again re-engineered at the point R2. Maintenance cost again comes down and software is ready to work for another life span. In this way, the process can be continued for long time. After delivery, software remains in the loop of the two stages maintenance and re-engineering till it retires as shown in the figure 4.1 (section 5.1.2). The legacy software is used for long time with this technique.
According to the above model software system is running in the forth life span and it is possible with re-engineering every time when it gets old. Right time for re-engineering is known by the decision making model(s) discussed earlier (section 5.2)

After re-engineering, the Software system will get another life span every time. This process can go for long time as software system never dies. It does not wear or tear out like other physical or living systems on the earth. It can be reengineered when it gets old. Question arises how many times to re-engineering the legacy system? When to replace the legacy system? The answer is when re-engineering is not possible and /or re-engineering cost is equal to new development cost of software of equal domain. We can change legacy system with new software when there is total change in our business process, technology, and/or environment. It is equivalent to say that when 100% objects are faulty, replace it with new development.

The idea behind this work is that re-engineering the software system after end of the first life span, then after the end of the 2nd life span, than after 3rd life span and so on as shown in the above figure 5.13. In the above model software is in the forth life span. Stop this process of re-engineering when there is big gap in business process and/or technology. Re-engineering cost will be equal to new development cost when business process and requirements are hundred percent changed.

It is beneficial to client companies because once purchased software can work for long time as legacy software. Software age can be increased manifolds by re-engineering when it gets aged. Software never depreciate, it never dies. When our requirements change or performance decreases or due to any other reason software do not meet our required output then it is called old or aged or obsolete. When software gets old, re-engineer it to reconstitute it to a new form. It will work for another life span. Again when the software gets old, re-engineer it for another life span. In this way the process continues till the re-engineering cost is less than the cost of new development of equal domain. It means software life can be increased many folds by re-engineering at the right time when it gets old. As re-engineering cost is significantly less than the new development cost, the software investment of
the client organizations can come down significantly. As shown in the above figure 5.13, Software system is in its fourth life span. Re-engineering is done third time, otherwise new software would have been purchased three times.

**Important features**

Once purchased software by the client company can work for long time. New software development can be escaped by re-engineering every time when software is not according to our requirements. Software investments can be decreased to significant levels.