Software Maintenance 9

Software maintenance is a task that every development group has to face when the software is delivered to the customer's site, installed and is operational. Therefore, delivery or release of software inaugurates the maintenance phase of the life cycle. The time spent and effort required keeping software operational after release is very significant and consumes about 40-70% of the cost of the entire life cycle. Despite the fact that it is very important and challenging task; it is routinely the poorly managed headache that nobody wants to do.

9.1 WHAT IS SOFTWARE MAINTENANCE

The term maintenance is a little strange when applied to software. In common speech, it means fixing things that break or wear out. In software, nothing wears out; it is either wrong from the beginning, or we decide later that we want to do something different. However, the term is so common that we must live with it [LAMB 88].

Software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization [STEP 78]. Because change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications. So any work done to change the software after it is in operation is considered to be maintenance work. The purpose is to preserve the value of software over time. The value can be enhanced by expanding the customer base, meeting additional requirements, becoming easier to use, more efficient and employing newer technology. Maintenance may span for 20 years, whereas development may be 1-2 years.

9.1.1 Categories of Maintenance

The only thing that remains constant in life is "CHANGE". As the specification of the computer systems change, reflecting changes in the external world, so must the systems themselves. More than two-fifths of maintenance activities are extensions and modifications requested by the users. There are four major categories of software maintenance, which are discussed below:

1. *Corrective maintenance*

This refers to modifications initiated by defects in the software. A defect can result from design errors, logic errors and coding errors. Design errors occur when changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test data. Coding errors are caused by incorrect

implementation of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors [LIEN 80].

In the event of system failure due to an error, actions are taken to restore operation of the software system. Due to pressure from management, maintenance personnel sometimes resort to emergency fixes known as "patching" [BENN 91]. The adhoc nature of this approach often gives rise to a range of problems that include increased program complexity and unforeseen ripple effects [TAKA 96]. Unforeseen ripple effects imply that a change to one part of a program may affect other sections in an unpredictable manner, thereby leading to distortion in the logic of the system. This is often due to lack of time to carry out a through "impact analysis" before effecting the change.

2. *Adaptive maintenance*

It includes modifying the software to match changes in the ever-changing environment. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the software, for example, business rules, government policies, work patterns, software and hardware operating platforms. A change to the whole or part of this environment will require a corresponding modification of the software [BROO 87].

Thus, this type of maintenance includes any work initiated as a consequence of moving the software to a different hardware or software platform-compiler, operating system or new processor. Any change in the government policy can have far-reaching ramifications on the software. When European countries had decided to go for "single European currency", this change affected all banking system software and was modified accordingly.

3. *Perfective maintenance*

It means improving processing efficiency or performance, or restructuring the software to improve changeability. When the software becomes useful, the user tends to experiment with new cases beyond the scope for which it was initially developed. Expansion in requirements can take the form of enhancement of existing system functionality or improvement in computational efficiency; for example, providing a Management Information System with a data entry module or a new message handling facility [STRI 82].

Hence, perfective maintenance refers to enhancements: making the product better, faster, smaller, better documented, cleaner structured, with more functions or reports.

4. *Other types of maintenance* Preventive

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflects deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance. This term is often used with hardware systems and implies such things as lubrication of parts before need occurs, or automatic replacement of banks of light bulbs before they start to individually burn out. Since software does not degrade in the same way as hardware and does not need maintenance to retain the presently established level of functionality. Some authors do not use this term. That is why we have included this type of activity under "Other Types of Maintenance" category.

This activity is usually initiated from within the maintenance organization with the intention of making program easier to understand and hence facilitating future maintenance work. This includes code restructuring, code optimization and documentation updating. After a series of quick fixes to software, the complexity of its source code can increase to an unmanageable level, thus justifying complete restructuring of the code. Code optimization can be performed to enable the programs to run faster or to make more efficient use of storage. Updating user and system documentation, though frequently ignored, is often necessary when any part of software is changed. The documents affected by the change should be modified to reflect the current state of the system [TAKA 96].

The requests come regularly to carry out maintenance activities. The request may be for corrective, adaptive or perfective maintenance. However most of the requests are for perfective maintenance. The distribution is shown in Fig. 9.1 [LIEN 80].

"Other types of maintenance" is required, as mentioned earlier, to reduce the complexity of the code. This is not very high as shown in Fig. 9.1, because, we may not get any external requests for this activity and this is confined to maintenance organization only.

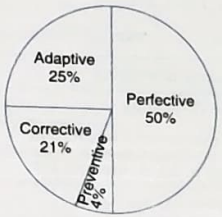

Fig. 9.1: Distribution of maintenance effort

9.1.2 Problems During Maintenance

The most important problem during maintenance is that before correcting or modifying a program, the programmer must first understand it. Then, the programmer must understand the impact of the intended change. Few problems are discussed below [LEHM 85, ARNO 82]:

CODE NOT READABLE
- Often the program is written by another person or group of persons working over the years in isolation from each other. Readability

CHANGES MISUND.
- Often the program is changed by person who did not understand it clearly, resulting in a deterioration of the program's original organization.

NOT STRUCTURED
- Program listings, even those that are well organized, are not structured to support reading for comprehension. We normally read an article or book straight through, but with listing, programmer rummages back and forth.

RIPPLE EFFECT
- There is a high staff turnover within information technology industry. Due to this many systems are maintained by persons who are not the original authors. These persons may not have adequate knowledge about the system. This may mean that these persons may introduce changes to programs without being aware of their effects on other parts of the system—the ripple effect. This problem may be worsened by the absence of documentation. Even where it exists, it may be out of date or inadequate.

- Some problems only become clearer when a system is in use. Many users know what they want but lack the ability to express it in a form understandable to programmers/ analysts. This is primarily due to information gap.
- Systems are not designed for change. If there is hardly any scope for change, maintenance will be very difficult. Therefore approach of development should be the production of maintainable software.

All these problems translate to a huge maintenance expenditure, which has been estimated ranging from 40%–70% of the total cost during the life cycle of large scale software systems [STEP 80].

9.1.3 Maintenance is Manageable

A common misconception about maintenance is that it is not manageable. We cannot control it. Every fix is different, and it is like hysterical phone calls at 2.00 in the morning.

The data obtained by Lientz and Swanson [LIEN 80] suggested other side of maintenance work. In their survey, they investigated the distribution of effort in software maintenance. The results are given in the Table 9.1.

Table 9.1: Distribution of maintenance effort

1	Emergency debugging	12.4%
2	Routine debugging	9.3%
3	Data environment adaptation	17.3%
4	Changes in hardware and OS	6.2%
5	Enhancements for users	41.8%
6	Documentation Improvement	5.5%
7	Code efficiency improvement	4.0%
8	Others	3.5%

It is clear from the Table 9.1, only 12.4% of effort is devoted to emergency debugging. Of course, getting a call at 2.00 in morning is more memorable than getting a request for an innocuous modification in a report format.

Another item in the survey of Lientz and Swanson asked about the kinds of maintenance requests that are made in the companies. These results are given in table 9.2.

Table 9.2: Kinds of maintenance requests

1	New reports	40.8%
2	Add data in existing reports	27.1%
3	Reformed reports	10%
4	Condense reports	5.6%
5	Consolidate reports	6.4%
6	Others	10.1%

The key lesson to learn from these studies is that most maintenance can be managed. Certainly the 79% in the adaptive, perfective and preventive categories can be anticipated, scheduled, monitored, estimated and managed. From the Table 9.1, we see that at least some of the corrective maintenance activities can also be managed. We may say, only about 10% of maintenance requests really require special and immediate handling. The other 90% requests can be handled in an organized and routine managerial system.

If we do proper and timely preventive maintenance, this 10% emergency requests may also be minimised. The preventive maintenance would involve assigning some time of maintenance persons to re-examine a product that has been modified several times to evaluate whether its structure can be reconstituted, cleaned up, or enhanced in anticipation of further maintenance. Preventive maintenance may cost, but it produces a real benefit to reduce the maintenance activities.

9.1.4 Potential Solutions to Maintenance Problems

A number of possible solutions to maintenance problems have been suggested. They include: budget and effort reallocation; complete replacement of existing systems; and enhancement of existing systems.

1. *Budget and effort reallocation*

It is now-a-days suggested that more time and resources should be invested in the development-specification and design of more maintainable systems rather than allocating resources to develop unmaintainable or difficult to maintain systems. The use of more advanced requirement specification approaches [BOTT 94], design techniques and tools [ZEVG 95], quality assurance procedures and standards such as ISOP 9000, CMM and maintenance standards are aimed at addressing this issue.

2. *Complete replacement of the system*

If maintaining an existing system costs as much as developing a new one, why not develop a new system from scratch. This point of view is understandable, but in practice it is not simple. The risk and costs associated with complete system replacement are very high. Corrective and preventive maintenance take place periodically at relatively small but incremental costs. Some organizations can afford to pay for these comparatively small maintenance charges while at the same time supporting more ambitious and financially demanding projects may not be possible.

The creation of another system is no guarantee that it will function better than the old one. On installation, errors may be found in functions which the old system performed correctly and such errors will need to be fixed as they are discovered.

3. *Maintenance of existing system*

Complete replacement of the system is not usually a viable option. An operational system in itself can be an asset to an organization in terms of the investment in technical knowledge and the working culture engendered. The current system may need to have the potential to evolve to a higher state, providing more sophisticated user-driven functionality, the capability of developing cutting edge technology, and of allowing the integration of other systems in a cost effective manner.

9.2 THE MAINTENANCE PROCESS

Once particular maintenance objective is established, the maintenance personnel must first understand what they are to modify. They must then modify the program to satisfy the maintenance objectives. After modification, they must ensure that the modification does not affect other portions of the program. Finally, they must test the program. These activities can be accomplished in the four phases as shown in Fig. 9.2 [STEP 80].

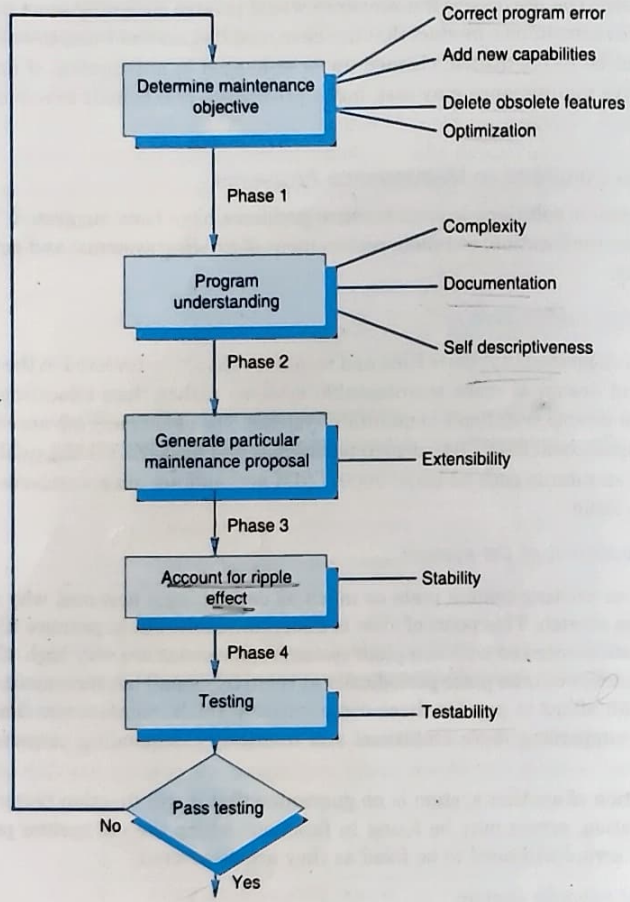

Fig. 9.2: The software maintenance process

9.2.1 Program Understanding

The first phase consists of analyzing the program in order to understand it. Several attributes such as the complexity of the program, the documentation, and the self-descriptiveness of the program contribute to the ease of understanding the program. The complexity of the program is a measure of the effort required to understand the program and is usually based on the control or data flow of the program. The self-descriptiveness of the program is a measure of how clear the program is, *i.e.*, how easy it is to read, understand, and use.

9.2.2 Generating Particular Maintenance Proposal

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified. However, the ease of generating maintenance proposals for a program is primarily affected by the attribute extensibility. The extensibility of the program is a measure of the extent to which the program can support extensions of critical functions.

9.2.3 Ripple Effect

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications. In software, the effect of a modification may not be local to the modification, but may also affect other portions of the program. There is a ripple effect from the location of the modification to the other parts of the programs that are affected by the modification [COLLO 78]. One aspect of this ripple effect is logical or functional in nature. Another aspect of this ripple effect concerns the performance of the program. Since a large-scale program usually has both functional and performance requirements, it is necessary to understand the potential effect of a program modification from both a logical and a performance point of view. The primary attribute affecting the ripple effect as a consequence of a program modification is the stability of the program. Program stability is defined as the resistance to the amplification of changes in the program.

9.2.4 Modified Program Testing

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before. It is important that cost-effective testing techniques be applied during maintenance. The primary factor contributing to the development of these cost-effective techniques is the testability of the program. Program testability is defined as a measure of the effort required to adequately test the program according to some well defined testing criterion.

9.2.5 Maintainability

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these factors must be combined to form maintainability. How easy is it to maintain a program? To a large extent, that depends on how difficult the program is to understand. Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain. And the more difficult it is to maintain, the higher its maintainability risk. Maintainability may be defined qualitatively as: the ease with which software can be understood, corrected, adapted, and/or enhanced

9.3 MAINTENANCE MODELS

Maintenance is nothing but development and requires special skills. Many times, maintenance activities are performed without requirements or design documents. Sometimes it is also difficult to understand the old code. Therefore, the need of maintenance models has been recognized, but presently, the models are neither so well developed nor so well understood as models for software development.

9.3.1 Quick-fix Model

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible. The model is shown in the Fig. 9.3.

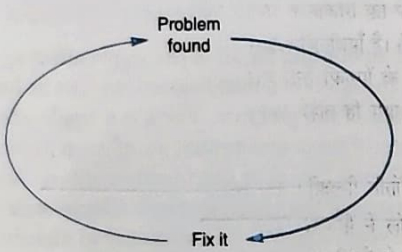

Fig. 9.3: The quick-fix model

In this model, fixes would be done without detailed analysis of the long-term effects, for example, ripple effects through the software or effects on the code structure. There would be little, if any documentation. Where are the advantages of such a model and why it is still used? In an appropriate environment it can work perfectly well. If, for example, a system is developed and maintained by a single person, he or she can come to learn the system well enough to be able to manage without detailed documentation, to be able to make instinctive judgements about how and how not to implement change. The job gets done quickly and cheaply. This is normally used due to pressure of deadlines and resources.

If customers are demanding the correction of an error they may not be willing to wait for the organization to go through detailed and time-consuming stage of risk analysis. The organization may run a higher risk in keeping its customers waiting than it runs in going for the quickest fix. But what of the long-term problems?

If an organization relies on quick-fix alone, it will run into difficult and very expensive problems, thus losing any advantage it gained from using the quick-fix model in the first place.

We should distinguish the short-term and long-term upgrades. If a user finds a bug in a commercial word processor, for example, it would be unrealistic to expect a whole new upgrade immediately. Often, a company will release a quick-fix as a temporary measure. The real solution will be implemented, alongwith other corrections and enhancements, as a major upgrade at a later date [TAKA 96].

9.3.2 Iterative Enhancement Model

In this model, it has been proposed that the changes in the software system throughout its lifetime are an iterative process. Originally proposed as a development model but well suited

to maintenance, the motivation for this was the environment where requirements were not fully understood and a full system could not be built.

Adapted for maintenance, the model assumes complete documentation as it relies on modification of this as the starting point of each iteration. The model is effectively a three-stage cycle as shown in the Fig. 9.4.

- Analysis
- Characterization of proposed modifications
- Redesign and implementation

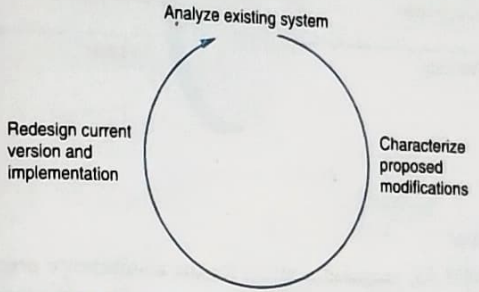

Fig. 9.4: The three-stage cycle of iterative enhancement

The existing documentation of each stage (requirements, design, coding, testing and analysis) is modified starting with the highest-level document affected by the proposed changes. These modifications are propagated through the set of documents and the system redesigned. The model explicitly supports reuse and also accommodates other models, for example the quick-fix model.

The pressure of the maintenance environment often dictates that a quick solution is found but, as we have seen, the use of the quickest solution can lead to more problems than it solves. In the first iteration, problem areas would be identified and next iteration would specifically address the problem.

The problems with this model are the assumptions made about the existence of full documentation and the ability of the maintenance team to analyze the existing product in full. Wider use of structured maintenance models will lead to a culture where documentation tends to be kept up to date and complete, the current situation is that this is not often the case.

9.3.3 Reuse Oriented Model

This model is based on the principle that maintenance could be viewed as an activity involving the reuse of existing program components. The reuse model [BASI 90] has four main steps:

(*i*) Identification of the parts of the old system that are candidates for reuse.
(*ii*) Understanding these system parts.
(*iii*) Modification of the old system parts appropriate to the new requirements.
(*iv*) Integration of the modified parts into the new system.

A detailed framework is required for the classification of components and the possible modifications. With the full reuse model the starting point may be any phase of the life cycle —

the requirements, the design, the code or the test data—unlike other models. (The model is shown in Fig. 9.5) For example, in the quick-fix model, the starting point is always the code.

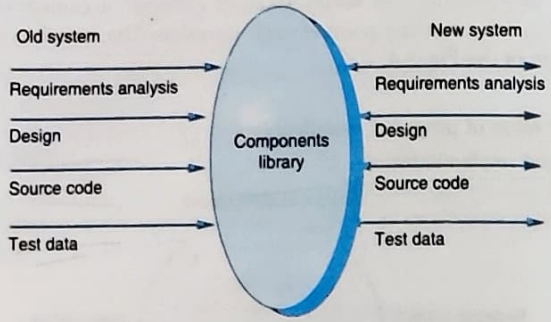

Fig. 9.5: The reuse model

9.3.4 Boehm's Model

In 1983 Boehm [BOEH 83] proposed a model for the maintenance process based upon the economic models and principles. Economic models are nothing new, economic decisions are a major driving force behind many processes and Boehm's thesis was that economic models and principles could not only improve productivity in the maintenance but also help understanding the process.

Boehm represents the maintenance process as a closed loop cycle as shown in Fig. 9.6. He theorizes that it is the stage where management decisions are made that drives the process. In this stage, a set of approved changes is determined by applying particular strategies and cost-benefit evaluations to a set of proposed changes. The approved changes are accompanied by their own budgets, which will largely determine the extent and type of resources expanded.

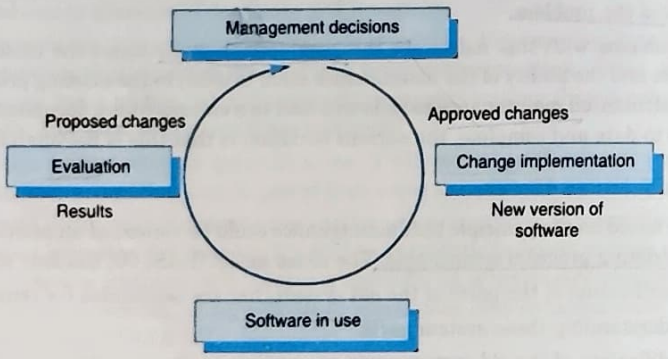

Fig. 9.6: Boehm's model

Boehm sees the maintenance manager's task as one of balancing the pursuit of the objectives of maintenance against the constraints imposed by the environment in which

maintenance work is carried out. Thus the maintenance process is driven by the maintenance manager's decisions, which are based on the balancing of objectives against the constraints.

9.3.5. Taute Maintenance Model

The model was developed by B.J. Taute in 1983 and is very easy to understand and implement. It is a typical maintenance model and has eight phases in cycle fashion. The phases are shown in Fig. 9.7.

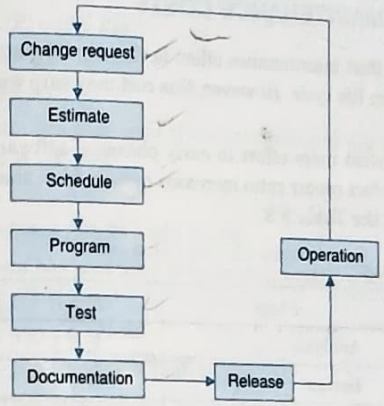

Fig. 9.7: Taute maintenance model

(i) **Change request phase:** Maintenance team gets a request in a prescribed format from the client to make a change. This change may fall in any category of maintenance activities. We identify the type of request (*i.e.* corrective, adaptive, perfective or preventive) and assign a unique identification number to the request.

(ii) **Estimate Phase:** This phase is devoted to estimate the time and effort required to make the change. It is difficult to make exact estimates. But our objective is to have at least reasonable estimate of time and effort. Impact analysis on existing system is also required to minimise the ripple effect.

(iii) **Schedule phase:** We may like to identify change requests for the next scheduled release and may also prepare the documents that are required for planning.

(iv) **Programming phase:** In this phase, source code is modified to implement the requested change. All relevant documents like design document, manuals etc. are updated accordingly. Final output is the test version of the source code.

(v) **Test phase:** We would like to ensure that modification is correctly implemented. Hence, we test the code. We may use already available test cases and may also design new test cases. The term used for such testing is known as regression testing.

(vi) **Documentation phase:** After regression testing, system and user documents are prepared/ updated before releasing the system. This helps us to maintain co-relation between code and documents.

(vii) **Release phase:** The new software product alongwith updated documents are delivered to the customer. Acceptance testing is carried out by the users of the system.

(*viii*) **Operation phase:** After acceptance testing, software is placed under normal operation. During usage, when another problem is identified or new functionality requirement is felt or enhancement of existing capability is desired, again a 'Change request' process is initiated. If we do so, we may go back to change requrest phase and repeat all phases to implement the change.

9.4 ESTIMATION OF MAINTENANCE COSTS

We had earlier discussed that maintenance effort is very significant and consumes about 40–70% of the cost of the entire life cycle. However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of software life cycle to reduce the maintenance costs. The defect repair ratio increases heavily from analysis phase to implementation phase and given in the Table 9.3.

Table 9.3: Defect repair ratio

Phase	*Ratio*
Analysis	1
Design	10
Implementation	100

Therefore more effort during development will certainly reduce the cost of maintenance. Good software engineering techniques such as precise specification, loose coupling and configuration management all reduce maintenance costs.

9.4.1 Belady and Lehman Model

This model indicates that the effort and cost can increase exponentially if poor software development approach is used and the person or group that used the approach is no longer available to perform maintenance. The basic equation [BELA 76] is given below:

$$M = P + K\, e^{(c-d)}$$

where

M : Total effort expended.

P : Productive effort that involves analysis, design, coding, testing and evaluation.

K : An empirically determined constant.

c : Complexity measure due to lack of good design and documentation.

d : Degree to which maintenance team is familiar with the software.

In this relation, the value of '*c*' is increased if the software system is developed without use of a software engineering process. Of course, '*c*' will be higher for a large software product with a high degree of systematic structure than a small one with the same degree. If the software is maintained without an understanding of the structure, function, and purpose of the software, then the value of '*d*' will be low [SAGE 90].

Example 9.1

The development effort for a software project is 500 person-months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if

(*i*) maintenance team has good level of understanding of the project ($d = 0.9$)

(*ii*) maintenance team has poor understanding of project ($d = 0.1$).

Solution

Development effort (P) = 500 PM

K = 0.3

C = 8

(*i*) Maintenance team has good level of understanding ($d = 0.9$)

$$M = P + K\, e^{(c-d)}$$
$$= 500 + 0.3\, e^{(8-0.9)}$$
$$= 500 + 363.59 = 863.59 \text{ PM}$$

(*ii*) Maintenance team has poor level of understanding ($d = 0.1$)

$$M = P + K\, e^{(c-d)}$$
$$= 500 + 0.3e^{(8-0.1)}$$
$$= 500 + 809.18 = 1309.18 \text{ PM}$$

Hence, it is clear that effort increases exponentially, if poor software engineering approaches are used and understandability of the project is poor.

9.4.2 Boehm Model

Boehm [BOEH81] proposed a formula for estimating maintenance costs as part of his COCOMO Model. Using data gathered from several projects, this formula was established in terms of effort. Boehm used a quantity called Annual Change Traffic (ACT) which is defined as:

"The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification".

The ACT is clearly related to the number of change requests.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

The Annual Maintenance Effort (AME) in person-months can be calculated as:

$$AME = ACT \times SDE$$

where, SDE : Software development effort in person-months.

ACT : Annual change Traffic

Suppose a software project required 400 person-months of development effort and it was estimated that 25% of the code would be modified in a year. The AME will be 0.25 x 400 = 100 person_month. Boehm suggested that this rough estimate should be refined by judging the importance of factors affecting the cost and selecting the appropriate cost multipliers. Using these factors, Effort Adjustment Factor (EAF) should be calculated as in the case of COCOMO model. The modified equation is given below:

$$AME = ACT * SDE * EAF$$

Example 9.2

Annual change traffic (ACT) for a software system is 15% per year. The development effort is 600 PMs. Compute an estimate for annual maintenance effort (AME). If life time of the project is 10 years, what is the total effort of the project?

Solution

The development effort = 600 PM

Annual change traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years.

The maintenance effort is a fraction of development effort and is assumed to be constant.

$$AME = ACT \times SDE$$
$$= 0.15 \times 600 = 90 \text{ PM}$$

Maintenance effort for 10 years $= 10 \times 90 = 900$ PM.

Total effort $= 600 + 900 = 1500$ PM.

Example 9.3

A software project has development effort of 500 PM. It is assumed that 10% code will be modified per year. Some of the cost multipliers are given as:

(*i*) Required software Reliability (RELY) : high

(*ii*) Date base size (DATA) : high

(*iii*) Analyst capability (ACAP) : high

(*iv*) Application experience (AEXP) : Very high

(*v*) Programming language experience (LEXP) : high

Other multipliers are nominal. Calculate the Annual Maintenance Effort (AME).

Solution

Annual change traffic (ACT) = 10%

Software development effort (SDE) = 500 PM

Using Table 4.5 of COCOMO model, effort adjustment factor can be calculated given below:

RELY = 1.15
ACAP = 0.86
AEXP = 0.82
LEXP = 0.95
DATA = 1.08

Other values are nominal values. Hence,

$$EAF = 1.15 \times 0.86 \times 0.82 \times 0.95 \times 1.08 = 0.832$$
$$AME = ACT * SDE * EAF$$
$$= 0.1 * 500 * 0.832 = 41.6 \text{ PM}$$
$$AME = 41.6 \text{ PM}$$

9.5 REGRESSION TESTING

Software inevitably changes, how so ever well written and designed it may be initially. This changed software is required to be retested in order to ensure that changes work correctly and these changes have not adversely affected other parts of the software. This is necessary because small changes in one part of a software may have subtle undesired effects in other seemingly unrelated parts of the software.

When we develop software, we use development testing to obtain confidence in the correctness of the software. Development testing involves constructing a test plan that describes how should we test the software, and then, designing and running suite of test cases that satisfy the requirements of the test plan. When we modify software, we typically retest it. This retesting is called regression testing. Hence, "Regression testing is the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code".

Therefore, regression testing tests both the modified code and other parts of the program that may be affected by the program change. It serves many purposes such as to:

- increase confidence in the correctness of the modified program
- locate errors in the modified program
- preserve the quality and reliability of software
- ensure the software's continued operation.

9.5.1 Development Testing Versus Regression Testing

We typically think of regression testing as a software maintenance activity; however, we also perform regression testing during the latter stages of software development. This latter stage starts after we have developed test plans and test suites and used them initially to test the software.

During this stage of development, we fine tune the code and correct errors in it, hence our activities resemble maintenance activities. The comparision of both is given in Table 9.4.

Table 9.4: Comparision of development and regression testing techniques

Sr. No.	*Development testing*	*Regression testing*
1.	We create test suites and test plans.	We can make use of existing test suites and test plans.
2.	We test all software components.	We retest affected components that have been modified by modifications.
3.	Budget gives time for testing.	Budget often does not give time for regression testing.
4.	We perform testing just once on a software product.	We perform regression testing many times over the life of the software product.
5.	Performed under the pressure of release date of the software.	Performed in crisis situations, under greater time constraints.

9.5.2 Regression Test Selection

Regression testing is very expensive activity and consumes significant amount of effort/cost. Many techniques are available to reduce this effort/cost. Simplest is to reuse the test suite that was used to test the original version of the software. Re-running all test cases in the test-suite, however, may still require excessive time.

An improvement is to reuse the existing test suite, but to apply a regression test selection technique to select an appropriate subset of the test suite to be run. If the subset is small enough, significant savings in time are achieved. To date, a number of regression test selection techniques have been developed for use in testing procedural and object oriented languages.

Testing professionals are reluctant, however, to omit from a test suite any test case that might expose a fault in the modified software. Consider, for example, the code fragments and associated test cases in Fig. 9.8 and Fig. 9.9 respectively. In Fig. 9.8, fragment B represents an erroneously modified version of fragment A, in which statement S_5 is modified.

We execute test cases t_3 and t_4, where both execute statements S_5 and S_5' Test case t_3 causes a divide by zero problem in S_5', whereas test case t_4 does not.

Fragment A		*Fragment B (modified form of A)*	
S_1	$y = (x - 1) * (x + 1)$	S_1'	$y = (x - 1) * (x + 1)$
S_2	if $(y = 0)$	S_2'	if $(y = 0)$
S_3	return (error)	S_3'	return (error)
S_4	else	S_4'	else
S_5	return $\left(\frac{1}{y}\right)$	S_5'	return $\left(\frac{1}{y-3}\right)$

Fig. 9.8: code fragments A and B

Test cases		
Test number	*Input*	*Execution history*
t_1	$x = 1$	S_1, S_2, S_3
t_2	$x = -1$	S_1, S_2, S_3
t_3	$x = 2$	S_1, S_2, S_5
t_4	$x = 0$	S_1, S_2, S_5

Fig. 9.9: Test cases for code fragment A of Fig. 9.8

If we execute all test cases, we will detect this divide by zero fault. But we have to minimize the test suite. From the Fig. 9.9, it is clear that test cases t_3 and t_4 have the same execution history *i.e.* S_1, S_2, S_5. If few test cases have the same execution history ; minimization methods select only one test case. Others may be selected for coverage elsewhere in the code. Hence, either t_3 or t_4 will be selected. If we select t_4, we lose the opportunity to expose the fault that t_3 exposes.

Hence minimization methods can omit some test cases that might expose fault in the modified software and so, they are not safe. We should be careful in the process of minimization of test cases and always try to use safe regression test selection technique.

A safe regression test selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program [ROTH96].

9.5.3 Selective Retest Techniques

Selective retest techniques differ from the "retest-all" technique, which reruns all tests in the existing test suite.

Selective retest technique may be more economical than the "retest-all" technique if the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that the selective retest technique lets us omit.

Selective retest techniques partition an existing test suite into subsets containing reusable, retestable, and obsolete test cases. Reusable test cases exercise only unmodified code and unmodified specifications, and do not need to run, but may be saved for reuse in subsequent testing sessions. Retestable test cases exercise modified code or test modified specification and should be repeated. Obsolete test cases either (*i*) specify incorrect input-output relations due to specification modifications or (*ii*) no longer exercise the program components or specifications they were designed to exercise. The test cases that are obsolete for the first reason are called specification-obsolete test cases, and the test cases that are obsolete for the second reason are called coverage-obsolete test cases. Because it may be difficult to recognize obsolete test cases, we may list some test cases as unclassified. In addition to reclassifying existing tests, selective retest techniques may create, or recommend creation of, new-structural or new-specification test cases, that test the program constructs or specification items which are not covered by existing test cases. Selective retest techniques are broadly classified in three categories :

1. Coverage techniques: They are based on test coverage criteria. They locate coverable program components that have been modified, and select test cases that exercise these components.

2. Minimization techniques: They work like coverage techniques, except that they select minimal sets of test cases.

3. Safe techniques: They do not focus on coverage criteria ; instead they select every test, case that causes a modified program to produce different output than its original version.

Rothermal [ROTH 96*a*] identified categories in which regression test selection techniques can be compared and evaluated. These categories are:

- inclusiveness
- precision
- efficiency
- generality.

Inclusiveness measures the extent to which a technique chooses test cases that will cause the modified program to produce different output than the original program, and thereby expose faults caused by modifications.

Precision measures the ability of a technique to avoid choosing test cases that will not cause the modified program to produce different output than the original program.

Efficiency measures the computational cost, and thus, practicality, of a technique.

Generality measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex modifications, and realistic testing applications.

Evaluation and comparision of existing techniques helps us to choose appropriate techniques for particular applications.

For example, if we require very reliable code, we may insist on a safe selective technique regardless of the cost. On the other hand, if we want to reduce the testing time, we may choose minimization technique, even though in doing so we may fail to select some test cases that expose faults in the modified program.

9.6 REVERSE ENGINEERING

Reverse engineering is the process followed in order to find difficult, unknown and hidden information about a software system. It is becoming important, since several software products lack proper documentation, and are highly unstructured, or their structure has degraded through a series of maintenance efforts. Maintenance activities cannot be performed without a complete understanding of the software system.

Apparently, understanding software looks a trivial task, but in the absence of any external documentation or clues, the task often becomes almost impossible.

9.6.1 Scope and Tasks

Since the main purpose of reverse engineering is to recover information from the existing code or any other intermediate documents, any activity that requires program understanding at any level may fall within the scope of reverse engineering. What is achieved as a result of any reverse engineering activity varies according to what is going to be done with the extracted information. The areas where reverse engineering is applicable include (but not limited to): (*i*) program comprehension, (*ii*) redocumentation and/or document generation (*iii*) recovery of design approach and design details at any level of abstraction, (*iv*) identifying reusable components, (*v*) identifying components that need restructuring, (*vi*) recovering business rules, and (*vii*) understanding high-level system description. Processes attempting to re-design, re-structure and enhance functionality of a system are not within the scope of reverse engineering.

Reverse engineering encompasses a wide array of tasks related to understanding and modifying software systems. This array of tasks can be broken into a number of classes. A few of these classes are briefly discussed below:

- Mapping between application and program domains: Computer programs are representations of problem situations from some application domain. The programs usually do not contain any hint about the problem. The task of the reverse engineer is to reconstruct the mapping from the application domain to the program domain as shown in Fig. 9.10.

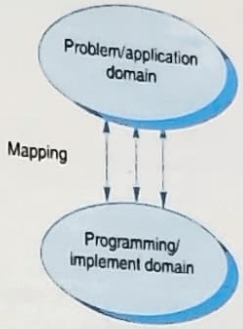

Fig. 9.10: Mapping between application and domains program

- Mapping between concrete and abstract levels: Software development process follows from high-level abstraction to more detailed design and concrete implementation. A reverse engineer has to move backward and create an abstract representation of the implementation from the mass of concrete details.
- Rediscovering high-level structures: A program is the embodiment of a well-defined purpose and coherent high-level structure. However, the purpose and structure may be lost in course of time and through maintenance activities, such as bug fixing, porting, modifying and enhancement. One of the tasks of reverse engineering is to detect the purpose and high-level structure of a program when the original one may have changed and where, in fact, there may be no such specific purpose left in the program.
- Finding missing links between program syntax and semantics: Computer programs are formal, in the sense they have well-defined syntax and semantics. In the formal world, the meaning of a syntactically correct program determines the output for a specific input. But systems that require reverse engineering generally would have lost their original semantics. Moreover, certain languages, such as the object-oriented languages, do not have strong formal basis. Reverse engineering process should determine the semantics of a given program from its syntax.
- To extract reusable component: Based on the premise that the use of existing program components can lead to an increase in productivity and improvement in product quality [BIGG 89], the concept of reuse has increasingly become popular amongst software engineers. Success in reusing components depends in part on their availability. Reverse engineering tools and methods offer the opportunity to access and extract program components.

9.6.2 Levels of Reverse Engineering

Reverse engineers detect low-level implementation constructs and replace them with their high level counterparts. The process eventually results in an incremental formation of an overall architecture of the program. It should, nonetheless, be noted that the product of a reverse engineering process does not necessarily have to be at a higher level of abstraction. If it is at the same level as the original system, the operation is commonly known as "redocumentation" [CHIK 90]. If on the other hand, the resulting product is at a higher level of

abstraction, the operation is known as "design recovery" [BIGG 89] or specification recovery as shown in Fig. 9.11 [TAKA 96].

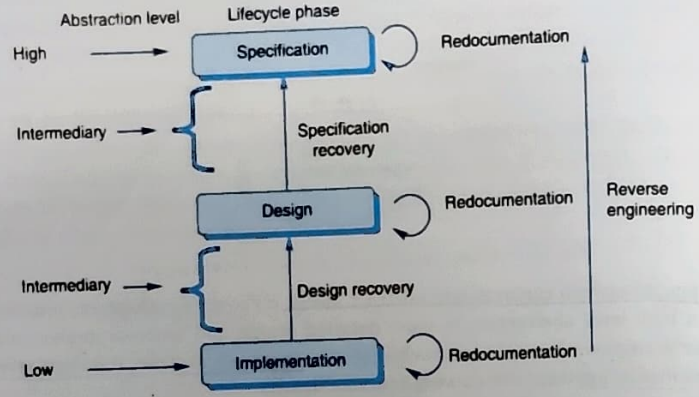

Fig. 9.11: Levels of abstraction

Redocumentation

Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level [CHIK 90]. The goals of this process are threefold. Firstly, to create alternative views of the system so as to enhance understanding, for example the generation of a hierarchical data flows or control flow diagram from source code. Secondly, to improve current documentation. Ideally, such documentation should have been produced during the development of the system and updated as the system changed. This, unfortunately, is not usually the case. Thirdly, to generate documentation for a newly modified program. This is aimed at facilitating future maintenance work on the system; preventive maintenance.

Design recovery

Design recovery entails identifying and extracting meaningful higher-level abstractions beyond those obtained directly from examination of the source code [CHIK 90]. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains [BIGG 89]. The recovered design—which is not necessarily the original design—can then be used for redeveloping the system. In other words, the resulting design forms a baseline for future system modifications [GILLI 90]. The design could also be used to develop similar but non-identical applications. For example, after recovering the design of a spelling check(er) application, it can be used in the design of a spell-checking module in a new word processing package.

Different approaches, which vary in their focus, can be used to recover these designs. Some draw heavily on programming language constructs contained in the program text as seen in the model by Rugaber et al. [RUGA 90]. They argue that an important aspect of design recovery is being able to recognize, understand and represent design decisions present in a given source code. Program constructs—which vary between programming languages—enable

recognition of design decisions. Examples of such constructs are control and data structures, variables, procedures and functions, definition and implementation modules, and class hierarchies.

9.6.3 Reverse Engineering Tools

Reverse engineering is performed with the help of certain tools, otherwise doing the job manually would require enormous amount of time and labour, rendering the whole purpose of it not only useless but also expensive. Rugaber [RUGA 94] identifies the following four basic components of a reverse engineering tool:

- The restructurer detects poorly structured code fragments and replaces them by equivalent structured code.
- The cross referencer lists the places where each variable is defined and used.
- The static analyzer detects anomalous constructs such as uninitialized variables and dead code.
- The text editor and other simple tools support browsing and editing of the source code.

In addition to these, a reverse engineering tool may incorporate graphical user-interface and other visualization tools. In the following subsections, we shall look at two reverse engineering tools.

RIGI

The Rigi System [MULL 92] is an interactive graph editor that provides a graphical representation of software systems. The Rigi approach to reverse engineering includes the following phases:

- Extracting system components and relationships from source code to create resource-flow graphs. The extraction phase is initially automatic and involves parsing the source code and storing the extracted artifacts (*i.e.*, datatypes, functions, dependencies) in a repository. The rest of the phases in Rigi are semi-automatic in the sense it requires user intervention.
- Creating subsystems from the flow-graphs using graph editor. Subsystem hierarchies are built on top of the initial call graphs by using the subsystem composition methodology, and subsystems are grouped into composite subsystems. These hierarchies can be documented as views, which are snapshots of the various reverse engineering states. Computing interfaces between subsystems by analyzing and propagating the dependencies extracted from the source code.
- Evaluating subsystems for cohesion and coupling and iterating until satisfactory subsystems have been created. Rigi is useful for identifying hot-spots for maintenance as well as candidate modules for reengineering. This information does not provide insight to the tool user on how to restructure the modules. The dynamic aspects of a software system are not modeled in Rigi and therefore it is doubtful how well it would perform for object-oriented languages.

"Refine" language tools

Reasoning systems, inc. markets the Refine Language Tools, a family of interactive, extensible workbenches for analyzing and reengineering code in programming languages including Ada, C, COBOL, and FORTRAN. One of the most attractive features of these tools is that they are customizable to suit any particular version of the programming language. Each refine language Tool comes with a fully documented reengineering API (application programming interface) for building customizations. Refine language tools use an intuitive X window system based graphical interface and provide capabilities including: (*i*) Interactive source code navigation, (*ii*) Generation of set/use, structure chart, and identifier definition reports, (*iii*) Online viewing and Postscript printing of all reports, (*iv*) Consistent graphical user interface and a standardized report format, and (*v*) Ability to export design information to forward-engineering CASE tools.

Reverse engineering plays a very significant role in software maintenance. Good tools can make the task of maintaining software less expensive and less troublesome. Experience and studies show that more than two-thirds of the money spent for a software system is chewed up by maintenance activities. In this backdrop, the role and importance of reverse engineering attains a very high standing. If a reverse engineering tool can reduce the maintenance cost by even a marginal amount, say 5%, it would save huge amount of money.

Since reverse engineering is applicable at any phase of software development or any level of abstraction, it can be effectively employed at strategic points during software development process. It would provide important feedback and enable the developers to go back, rectify faults, and design and implement more efficient software. Although this would increase development cost but eventually it would save much more from maintenance expenses.

9.7 SOFTWARE RE-ENGINEERING

Systems that have been in the field for a long time evolve and mutate in ways that were never planned. As enhancements are added, bugs fixed, and piece-meal solutions tacked on, the once elegant system grows into something unmanageable and expensive to maintain.

These old systems that must still be maintained are sometimes called legacy systems. These legacy systems may have been the best that technology had to offer. But, as mission requirements change and better technology becomes available, the users are faced with the difficult problem of reconciling the large investments they have already made in their existing systems against the promise of better designs, lower cost of maintenance, cheaper technology and large improvements in capabilities. Most of the legacy systems may be poorly structured and their documentation may be either out of date or non-existent. The developers of these systems having left the organization; there may be no one in the organization who really understands these systems in detail. These systems were also not designed for change. Another problem is the non-availability of requirements, design and test cases.

Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable. As a part of this re-engineering process, the system may be redocumented or restructured. It may be translated to a more modern programming language, implemented on existing hardware technology. Thus, software re-engineering allows us to translate source code to a new language, restructure our old code,

migrate to a new platform (such as client-server), capture and then graphically display design information, and re-document poorly documented systems.

The critical distinction between re-engineering and new software development is the starting point for the development as shown in Fig. 9.12. Rather than start with a written specification, the old system acts as a specification for the new system.

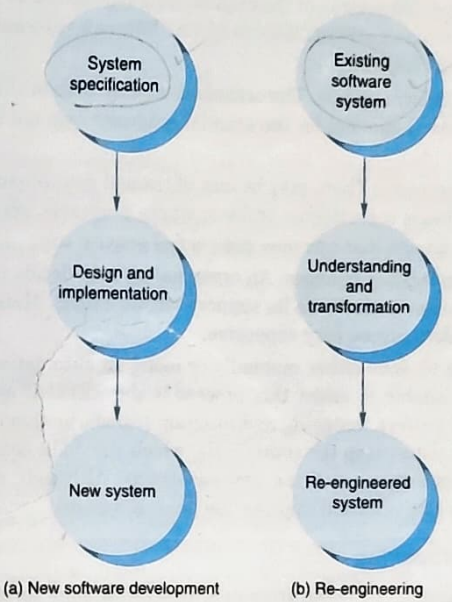

Fig. 9.8: Comparision of new software development with re-engineering

The costs of re-engineering depend on the extent of the work that is carried out. Other factors affecting costs are; the quality of the software, tool support available, extent of data conversion, availability of expert staff.

The alternative to re-engineering a software system is to redevelop that system using modern software engineering techniques. Where systems are very badly structured this may be the only viable option as the re-engineering costs for these systems are likely to be high.

The following suggestions may be useful for the modification of the legacy code:

- Study code well before attempting changes.
- Concentrate on overall control flow and not coding.
- Heavily comment internal code.
- Create cross references
- Build symbol tables
- Use own variables, constants and declarations to localize the effect of change.

- Keep detailed maintenance document.
- Use modern design techniques.

9.7.1 Source Code Translation

Simplest method is to translate source code to another programming language. The target language may be an updated version of the original language (Basic to Visual Basic) or may be a completely different language (FORTRAN to JAVA). Source level translation may be required for the following reasons [SOMM00]:

(*i*) Hardware platform update: The organization may wish to change its standard hardware platform. Compilers for the original language may not be available on the new platform.

(*ii*) Staff skill shortages: There may be lack of trained maintenance staff for the original language. This is a particular problem where programs were written in some non-standard language that has now gone out of general use.

(*iii*) Organizational policy changes: An organization may decide to standardize on a particular language to minimize its support software costs. Maintaining many versions of old compilers can be very expensive.

Translation can be done either manually or using an automation tool. One of the most sophisticated tools available to assist this process is the REFINE system [MARK 94] that incorporates powerful pattern matching and program transformation capabilities. It is therefore possible to define patterns in the source code, which should be converted. REFINE recognizes these and replaces them with the new constructs. Although, manual intervention is almost always required to tune and improve the general system.

9.7.2 Program Restructuring

This involves transforming a system from one representational form to another without a change in the semantics or functionality. The transformation would usually be to a more desirable format.

Due to maintenance activities, the programs become complex and difficult to understand. To control this increase in complexity, the source code needs to be restructured. There are various types of restructuring techniques and some are discussed below:

(*i*) **Control flow driven restructuring:** This involves the imposition of a clear control structure within the source code and can be either inter modular or intra-modular in nature. This makes the program more readable and easier to understand. However, the program may still suffer from a lack of modularity whereby related components of the program may be dispersed through the code.

(*ii*) **Efficiency driven restructuring:** This involves restructuring a function or algorithm to make it more efficient. A simple example is the replacement of an IF-THEN-ELSE-IF-ELSE construct with a CASE construct as shown in Fig. 9.13. With the CASE statement, only one boolean variable is evaluated, whereas with the IF-THEN-ELSE-IF-ELSE construct, more than one of the boolean expressions may need to be tested during execution thereby making it less efficient.


```
IF Score > = 75 THEN Grade: = 'A'
ELSE IF Score > = 60 THEN Grade: = 'B'
ELSE IF Score > = 50 THEN Grade: = 'C'
ELSE IF Score > = 40 THEN Grade: = 'D'
ELSE IF Grade = 'F'
END
```

(*a*)

```
CASE Score of
75, 100: Grade: = 'A'
60, 74: Grade: = 'B';
50, 59: Grade: = 'C';
40, 49: Grade: = 'D';
ELSE Grade: = 'F'
END
```

(*b*)

Fig. 9.13: Restructuring a program

(*iii*) **Adaption-driven restructing:** This involves changing the coding style in order to adapt the program to a new programming language or new operating environment, for instance changing an imperative program in PASCAL into a functional program in LISP. Another example is the transformation of a program functions in a sequential environment to an equivalent but totally different form of processing in a parallel environment.

9.8 CONFIGURATION MANAGEMENT

The software may be considered as configurations of software components. These software components are released in the form of executable code whereas supplier organization keeps the source code. This source code is the representation of an executable equivalent, but which one? Source code can be modified without there being any effect upon executable versions in use and, if strict controls are not kept, the source code which is the exact representation of a particular executable version may no longer exist. The means by which the process of software development and maintenance is controlled is called configuration management. The configuration management is different in development and maintenance phases of life cycle due to different environments. Software maintenance is undertaken in the environment of a live system in use by a probably large user base. Potential effects upon a live system are much more immediate than those upon a system still under development. Thus, configuration management is concerned with the development of procedures and standards for cost effective managing and controlling changes in an evolving software system.

9.8.1 Configuration Management Activities

The activities are divided into four broad categories [TAKA 96].

1. The identification of the components and changes.
2. The control of the way by which the changes are made.
3. Auditing the changes.
4. Status accounting-recording and documenting all the activities that have taken place.

The following documents are required for these activities:

- Project plan
- Software requirements specification document

- Software design description document
- Source code listing
- Test plans/procedures/test cases
- User manuals

All components of the system's configuration are recorded alongwith all relationships and dependencies between them. Any change-addition, deletion or modification-must be recorded and its effect upon the rest of the system's components should be checked. After a change has been made, a new configuration is recorded. There is a need to know who is responsible for every procedure and process along the way and it is a management task both to assign these responsibilities and to conduct audits to see that they are carried out.

9.8.2 Software Versions

During software maintenance, there will be at least two versions of the software system; the old version and the new version (s). Since a software system is comprised of software components, there will also be two or more versions of each component that has been changed. Thus, the software maintenance team has to cope with multiple versions of the software. We can distinguish between two types of versions namely revisions (replace) and variations (variety).

Version Control: During the process of software evolution, many objects are produced, for example files, electronic documents, paper documents, source code, executable code and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This comprises the

- Name of each source code component, including the variations and revisions,
- The versions of the various compilers and linkers used,
- The name of the software staff who constructed the component.
- The date and the time at which it was constructed.

9.8.3 Change Control Process

It will be appropriate if changes to software can be predicted. Change control process comes into effect when the software and associated documentation are delivered to configuration management change request form (as shown in Fig. 9.14), which should record the recommendations regarding the change. The recommendations may include assessment of the proposed change, the estimated costs and how the change should be implemented. This form is submitted to a Change Control Authority (CCA), which decides whether or not the change is to be accepted. If change is approved by the CCA, it is applied to the software. The revised software is revalidated by the Software Quality Assurance (SQA) team to ensure that the change has not adversely affected other parts of the software. The changed software is handed over to the software configuration team and is incorporated in a new version of the system.

CHANGE REQUEST FORM

Project ID:

Change Requester with date:

Requested change with date:

Change analyzer:

Components affected:

Associated components:

Estimated change costs:

Change priority:

Change assessment:

Change implementation:

Date submitted to CCA:

Date of CCA decision:

CCA decision:

Change implementer:

Date submitted to QA:

Date of implementation:

Date submitted to CM:

QA decision:

Fig. 9.14: Change request form

9.9 DOCUMENTATION

Software documentation is the written record of facts about a software system recorded with the intent to convey purpose, content and clarity. The recording process usually begins when the need for the system is conceived and continues until the system is no longer in use.

There are different categories of software documentation like user documentation, system documentation, etc.

9.9.1 User Documentation

It refers to those documents, containing descriptions of the functions of a system without reference to how these functions are implemented. A list of user documentation is given in Table 9.5.

Table 9.5: User documentation

S. No.	Document	Function
1.	System Overview	Provides general description of system's functions
2.	Installation Guide	Describes how to set up the system, customize it to local hardware needs and configure it to particular hardware and other software systems.
3.	Beginner's Guide	Provides simple explanations of how to start using the system.
4.	Reference Guide	Provides in-depth description of each system facility and how it can be used.
5.	Enhancement	Booklet contains a summary of new features.
6.	Quick reference card	Servers as a factual lookup.
7.	System administration	Provides information on services such as net-working, security and upgrading.

9.9.2 System Documentation

It refers to those documentation containing all facets of system, including analysis, specification, design, implementation, testing, security, error diagnosis and recovery. System documentation details are given in Table 9.6.

9.9.3 Other Classification Schemes

There are other ways in which documentation may be classified. There may be three levels of documentation: user manuals, operator's manuals and maintenance manuals. User manual describes what the system does without necessarily going into the details of how it does it or how to get the system to do it. The operator's manual describes how to use the system as well as giving instructions on how to recover from faults. The maintenance manual contains details of the functional specification, design, code listing, test data and results.

Table 9.6: System documentation

S. No.	Document	Function
1.	System Rationale	Describes the objectives of the entire system.
2.	SRS	Provides information on exact requirements of system as agreed between user and developer
3.	Specification/Design	Provides description of: (*i*) How system requirements are implemented. (*ii*) How the system is decomposed into a set of interacting program units. (*iii*) The function of each program unit.
4.	Implementation	Provides description of: (*i*) How the detailed system design is expressed in some formal programming language. (*ii*) Program actions in the form of intra program comments.

(Contd.)...

S. No.	Document	Function
5.	System Test Plan	Provides description of how program units are tested individually and how the whole system is tested after integration
6.	Acceptance Test Plan	Describes the tests that the system must pass before users accept it.
7.	Data Dictionaries	Contains description of all terms that relate to the software system in question.

This classification and user/system documentation schemes are identical in the sense that they both include all the information that is contained in software documents.

REFERENCES

[ARNO 82] Arnold R.S., "*The Dimensions of Healthy Maintenance*", IEEE Software Engineering, 1982.

[BASI 90] Basili V.R., "*Viewing Software Maintenance as Reuse–Oriented Software Development*", IEEE Software, 7, January, 19–25, 1990.

[BEIZ 90] Beizer B., "*Software Testing Techniques*", Van Nostrand Reinhold, New York NY, 1990.

[BELA 76] Belady L. and Lehman W., "*A Model of Large Program Development*", IBM Systems Journal, Vol. 15, No. 3, 225–252, 1976.

[BENN 91] Bennett K. et al., "*Software Maintenance*", Butterworth–Heinemann Ltd., Oxford, 1991.

[BIGG 89] Biggerstaff T.J., "*Design Recovery for Maintenance and Reuse*", Computer, 22(7), July, 36–49, 1989.

[BOEH 81] Boehm B.W., "*Software Engineering Economics*", Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[BOEH 83] Boehm B.W., "*The Economic of Software Maintenance*", Proc. Workshop on Software Maintenance", Silver Spring, MD, IEEE Computer Society Press, 9–37, 1983.

[BOTT 94] Bottaci L. and Jones J.G., "*Formal Specification on Using Z: A Modeling Approach*", Int. Thomson Publishing, London, 1994.

[BROO 87] Brooks R., "*No Silver Bullet–Essence and Accidents of Software Engineering*", IEEE Computer, 20 (4), 10–20, 1987.

[CHIK 90] Chikofsky E.J. and Cross Jr. J.H., "*Reverse Engineering and Design Recovery: A Taxonomy*", IEEE Software, 7, January, 13–17, 1990.

[COLLO 78] Collofello J.S. et al.,"*Ripple Effect Analysis of Software Maintenance*", Proc. COMPSAC 78, 60–65, 1978.

[GILLI 90] Gillis K.D. and Wright D.G., "*Improving Software Maintenance using System Level Reverse Engineering*", In Proc. IEEE Conference on Software Maintenance", 84–90, LA, IEEE Computer Society Press, 1990.

[LAMB 88] Lamb D.A., "*Software Engineering: Planning for Change*", Prentice Hall, Englewood Cliffs, NJ, 1988.

[LEHM 85] Lehman M.M., "*Program Evolution*", Academic Press, London, 1985.

[LIEN 80] Lientz B.P. and Swanson E.B., "*Software Maintenance Management*", Addison–Wesley Publishing Company, Reading, Massachusetts, 1980.

[MARK 94] Morkosian L. et al. "*Using an Enabling Technology to Reengineer Legacy Systems*", Communications of the ACM, 34(5), 58–70, May, 1994.

[MULL 92] Muller H.A. et al., "*A Reverse Engineering Environment Based on Spital and Visual Software Interconnection Models*", Software Engineering Notes, 17(5), Dec, 34, 88–98, 1992.

[ROTH 96] Rothermel G. and Harrold M.J., "*Analyzing Regression Test Selection Techniques*", IEEE Transactions on Software Engineering 22 (8), 529–551, August 1996.

[ROTH 96 *a*] Rothermel R., "*Efficient Effective Regression Testing Using Safe Test Selection Techniques*", Ph. D Thesis, Clemson University, May, 1996.

[RUGA 90] Rugaber S. et al., "*Recognizing Design Decisions in Programs*," IEEE Software, 7(1), January, 46–54, 1990.

[RUGA 94] Rugaber S., "*White Paper on Reverse Engineering*", Tech Report, Georgia Institute of Technology, March, 1994.

[SAGE 90] Sage A. and Palmer J.D., "*Software Systems Engineering*", John Wiley and Sons, 1990.

[SOMM 00] Sommerville Ian, "*Software Engineering*", Addison–Wesley, 2000.

[STEP 78] Stephen S.Y. et al., "*Ripple Effect Analysis of Software Maintenance*", in Proc. COMPSAC78, 1978.

[STEP 80] Stephen S.Y. and Collofello J.S., "*Some Stability Measures for Software Maintenance*", IEEE Software Engg., 1980.

[STEP 80] Stephen S.Y. and Collofello J.S., "*Some Stability Measures for Software Maintenance*", IEEE Trans on Software Engineering, 28–35, 1980.

[STRI 82] Strickland J.P. et al., "*An Evolving System*", IBM Systems Journal, 21 (4), 490–513, 1982.

[TAKA 96] Takang A.A., Grubb P.A., "*Software Maintenance*", Thomson Computer Press, U.K.

[TAUT 83] Taute B.J., "*Quality Assurance and Maintenance Application Systems*", Proc. of the National AFIPS computer conference, PP 123–129, 183.

[ZVEG 95] Zvegintzov N., "*Software Management Technology Reference Guide: 1994/95 European Edition*", Software Maintenance News, Inc., Los Alamitos, California, 1995.

MULTIPLE CHOICE QUESTIONS

Note: *Choose most appropriate answer of the following questions:*

9.1. Process of generating analysis and design documents is called
(*a*) inverse Engineering (*b*) software Engineering
(*c*) reverse Engineering (*d*) re-engineering.

9.2. Regression testing is primarily related to
(*a*) functional testing (*b*) data flow testing
(*c*) development testing (*d*) maintenance testing.

9.3. Which one is not a category of maintenance ?
(*a*) corrective maintenance (*b*) effective maintenance
(*c*) adaptive maintenance (*d*) perfective maintenance.

9.4. The maintenance initiated by defects in the software is called
(*a*) corrective maintenance (*b*) adaptive maintenance
(*c*) perfective maintenance (*d*) preventive maintenance.

9.5. Patch is known as
(*a*) emergency fixes (*b*) routine fixes
(*c*) critical fixes (*d*) none of the above.

9.6. Adaptive maintenance is related to
(*a*) modification in software due to failures
(*b*) modification in software due to demand of new functionalities
(*c*) modification in software due to increase in complexity
(*d*) modification in software to match changes in the ever-changing environment.

9.7. Perfective maintenance refers to enhancements
(*a*) making the product better
(*b*) making the product faster and smaller
(*c*) making the product with new functionalities
(*d*) all of the above.

9.8. As per distribution of maintenance effort, which type of maintenance has consumed maximum share?
(*a*) adaptive (*b*) corrective
(*c*) perfective (*d*) preventive.

9.9. As per distribution of maintenance effort, which type of maintenance has consumed minimum share?
(*a*) adaptive (*b*) corrective
(*c*) perfective (*d*) preventive.

9.10. Which one is not a maintenance model?
(*a*) CMM (*b*) iterative Enhancement model
(*c*) quick-fix model (*d*) reuse-oriented model.

9.11. In which model, fixes are done without detailed analysis of the long-term effects?
(*a*) reuse oriented model (*b*) quick-fix model
(*c*) taute maintenance model (*d*) none of the above.

9.12. Iterative enhancement model is a
(*a*) three stage model (*b*) two stage model
(*c*) four stage model (*d*) seven stage model.

9.13. Taute maintenance model has
(*a*) two phases (*b*) six phases
(*c*) eight phases (*d*) ten phases.

9.14. In Boehm model, ACT stands for
(*a*) actual change time (*b*) actual change traffic
(*c*) annual change traffic (*d*) annual change time.

9.15. Regression testing is known as
(*a*) the process of retesting the modified parts of the software
(*b*) the process of testing the design documents
(*c*) the process of reviewing the SRS
(*d*) none of the above.

9.16. The purpose of regression testing is to
(*a*) increase confidence in the correctness of the modified program
(*b*) locate errors in the modified program
(*c*) preserve the quality and reliability of software
(*d*) all of the above.

9.17. Regression testing is related to
(*a*) maintenance of software (*b*) development of software
(*c*) both (*a*) and (*b*). (*d*) none of the above.

9.18. Which one is not a selective retest technique
(a) coverage technique (b) minimization technique
(c) safe technique (d) maximization technique.

9.19. Purpose of reverse engineering is to
(a) recover information from the existing code or any other intermediate document
(b) redocumentation and/or document generation
(c) understand the source code and associated documents
(d) all of the above.

9.20. Legacy systems are
(a) old systems (b) new systems
(c) undeveloped systems (d) none of the above.

9.21. User documentation consists of
(a) system overview (b) installation guide
(c) reference guide (d) all of the above.

9.22. Which one is not a user documentation?
(a) beginner's Guide (b) installation guide
(c) SRS (d) system administration.

9.23. System documentation may not have
(a) SRS (b) design document
(c) acceptance Test Plan (d) system administration.

9.24. The process by which existing processes and methods are replaced by new techniques is :
(a) reverse engineering (b) business process re-engineering
(c) software configuration management (d) technical feasibility.

9.25. The process of transforming a model into source code is
(a) reverse engineering (b) forward engineering
(c) re-engineering (d) restructuring.

EXERCISE

9.1. What is software maintenance? Describe various cetegories of maintenance. Which category consumes maximum effort and why?

9.2. What are the implications of maintenance for a one person software production organization?

9.3. Some people feel that "maintenance is manageable". What is your opinion about this issue?

9.4. Discuss various problems during maintenance. Describe some solutions to these problems.

9.5. Why do you think that the mistake is frequently made of considering software maintenance inferior to software development?

9.6. Explain the importance of maintenance. Which category consumes maximum effort and why?

9.7. Explain the steps of software maintenance with help of a diagram.

9.8. What is self descriptiveness of a program? Explain the effect of this parameter on maintenance activities.

9.9. What is ripple effect? Discuss the various aspects of ripple effect and how does it affect the stability of a program?

9.10. What is maintainability? What is its role during maintenance?

9.11. Describe quick-fix model. What are the advantages and disadvantages of this model?

9.12. How iterative enhancement model is helpful during maintenance? Explain the various stage cycles of this model.

9.13. Explain the Boehm's maintenance model with the help of a diagram.

9.14. State the various steps of reuse oriented model. Is it a recommended model in object oriented design?

9.15. Describe the Taute maintenance model. What are various phases of this model?

9.16. Write a short note on Belady and Lehman model for the calculation of maintenance effort.

9.17. Describe various maintenance cost estimation models.

9.18. The development effort for a project is 600 PMs. The empirically determined constant (K) of Belady and Lehman model is 0.5. The complexity of code is quite high and is equal to 7. Calculate the total effort expended (M) if maintenance team has resonable level of understanding of the project ($d = 0.7$).

9.19. Annual change traffic (ACT) in a software system is 25% per year. The initial develpment cost was Rs. 20 lacs. Total life time for software is 10 years. what is the total cost of the software system?

9.20. What is regression testing? Differentiate between regression and development testing.

9.21. What is the importance of regression test selection? Discuss with the help of examples.

9.22. What are selective retest techniques? How are they different from "retest-all" technique?

9.23. Explain the various categories of retest techniques. Which one is not useful and why?

9.24. What are the categories to evaluate regression test selection techniques? Why do we use such categorisation?

9.25. What is reverse engineering? Discuss levels of reverse engineering.

9.26. What are the appropriate reverse engineeing tools? Discuss any two tools in detail.

9.27. Discuss reverse engineering and re-engineering.

9.28. What is re-engineering? Differentiate between re-engineering and new development.

9.29. Discuss the suggestions that may be useful for the modification of the legacy code.

9.30. Explain various types of restructuring techniques. How does restructuring help in maintaining a program?

9.31. Explain why single entry, single exit modules make testing easier during maintenance.

9.32. What are configuration management activities? Draw the performa of change request form.

9.33. Explain why the success of a system depends heavily on the quality of the documentation generated during system development.

9.34. What is an appropriate set of tools and documents required to maintain large software product?

9.35. Explain why a high degree of coupling among modules can make maintenance very difficult?

9.36. Is it feasible to specify maintainbility in the SRS? If yes, how would we specify it?

9.37. What tools and techniques are available for software maintenance? Discuss any two of them.

9.38. Why is maintenance programming becoming more challenging than new development? What are desirable characteristics of a maintenance programmer ?

9.39. Why little attention is paid to maintainability during design phase?

9.40. List out system documentation and also explain their purpose.