Software Maintenance

Chapter 10:

Management and Organisational issues

Large and complex software systems are the ones that present challenges for management because:

1. They form an integral part of an organisation,
2. Their ability to evolve is at the heart of their operation,
3. and their maintenance requires the services of large numbers of personnel.

It is the job of management to enable maintenance personnel to live up to this expectation.

Management Responsibilities

Management has the responsibility of ensuring that the software system under maintenance is of a satisfactory quality, and that desired changes are affected with the minimum possible delay at the least possible cost. This can be achieved by:

Devising a means of managing maintenance personnel in order to increase their productivity, ensure job satisfaction and improve system quality, all of which can be facilitated through choice of personnel, motivation, a suitable team structure and education and training;

Selecting a suitable way of organising maintenance tasks to increase productivity, control maintenance effort and cost, and most importantly deliver a high-quality system. This depends very much on the organisational modes employed for maintenance tasks.

Enhancing Maintenance Productivity

There are several ways in which this can be done. It is a management task to find the right people for the job, then to see that they are motivated, and given the necessary information and resources to do the job well.

1. Choosing the right People:

This means tackling the problem of low status of maintenance personnel and working towards improving the general image of maintenance work, an image not enhanced by the low status and low financial rewards traditionally given to maintenance staff. Another way of attracting high calibre people to maintenance work is by tying the overall aims of the organisation to the aims of maintenance.

2. Motivating Maintenance Personnel

A team should never be solely reliant on one or two "stars" whose loss would cause the enterprise to collapse.

A motivated team is far more likely to stay. The up-front costs of employing new experienced personnel may be greater, but the long-term investment will be worthwhile. Some ways of motivating personnel are:

a. Rewards:

It is important that maintenance personnel feel valued and rewarded for their hard work. One intuitively thinks of reward in financial terms - the one-off bonus payment for a particularly good piece of work. However, ill-throughout bonus schemes can be counter-productive. A structured reward system such as promotion is often more effective. If someone is capable of good enough work to earn bonus payments, he or she is likely to be someone worth retaining in a maintenance team. Promotion brings not only financial reward, but also enhanced status

b. Supervision:

It is not always possible to have highly experienced maintenance staff. There is a need to assign inexperienced staff to maintenance tasks. It is essential to ensure that they get the right level of

technical supervision and support from senior members of staff, otherwise they become demotivated through attempts to accomplish tasks beyond their skills and expertise.

c. Assignment Patterns:

From time to time, it is important to rotate maintenance personnel between maintenance and development tasks. One advantage is that they do not feel stuck with the maintenance tasks. Secondly, their maintenance experience can help them develop more maintainable systems.

d. Recognition:

Getting proper acknowledgement for the very real benefits that maintenance brings to an organisation can assist maintainers to recognise their importance within the organisation.

e. Career Structure:

Providing an effective maintenance career structure equivalent to that for development will help to engender a culture where maintenance is seen as a valuable activity by staff and management. It is important also that maintenance is viewed as an activity that enhances the assets of a company and not just as a drain on resources.

3. Communication:

It is important that management keep maintenance personnel informed. If staff are unaware of what the procedures are, they will be unable to follow them effectively or give competent feedback on problems associated with them. Information must flow in both directions in order that the maintenance process may be properly controlled, and information may be gathered on the processes carried out so that benefits may be documented and quantified.

Framework needed for effective communication:

1. Adequate resources:

Resources can be viewed in terms of tools - software and hardware - and working environment. It is important that investment in tools for maintenance is not secondary to investment in tools for development as this gives a clear signal of the perceived relative status of the two activities. In a study of maintenance programmers, they put state-of-the-art software tools at the top of their list of things most likely to increase productivity.

The issue of investing enough in maintenance staff also includes employing the appropriate number of staff. Over-stretching too few good people can lead to dissatisfaction with the working environment.

2. Domain Knowledge:

Managers, in order to be effective, must have adequate knowledge of the maintenance process. They need to be aware of the cost implications of the various maintenance stages in order to be able to guide the maintenance process effectively.

Maintenance Teams:

The structure of the maintenance team itself is an important factor in determining the level of productivity. Two types of teams commonly used in development are egoless programming and the chief programmer team.

1. Egoless Programming Team:

The egoless programming team is an organisational structure of individuals whose operation is based on the philosophy that everyone involved in a project should work together to develop the best possible software system. It requires a collegiate setting where team members are open to criticism and do not allow their ego to undermine the project objective.

2.Chief Programmer Team:

The chief programmer team imposes an organisational structure in which discipline, clear leadership and functional separation play a major role.
Its objectives are to organise software maintenance into clearly-defined tasks, to create an environment that promotes the use of state-of-the-art tools, and to ensure that at least two people understand every line of code. It differs sharply from egoless programming in the lack of a comparable level of democracy.

The major difficulty is that software maintenance tends to be change-driven without the overall timetable and budget considerations given to development projects. The maintenance process is initiated when there is a change to be made. This change may be too small to warrant the services of a team. At other times though, there could be a request for a major maintenance task thereby justifying a team effort. To address the differences, Martin and McClure have suggested two types of maintenance team: the **short-term (temporary) team** and the **long-term (permanent) team**.

Temporary Team:

A temporary team is created on an informal basis when there is a need to perform a specific task.
For Ex: Code Review. The programmers work together to solve the problem at hand.
Leadership is not fixed; it rotates between the team members.
The main problem with this arrangement is that program quality, programmer morale and user satisfaction can be compromised.

Permanent Team:

A permanent team is a more formal arrangement. It allows for specialisation, creates communication channels, promotes an egoless, collegiate atmosphere, reduces dependency on individuals and allows for periodic audit checks. This team is created on a permanent basis to oversee the successful evolution of a software system throughout its lifetime. The team consists of a maintenance leader, a co-leader, a user-liaison person, a maintenance administrator, and other programmers.

The maintenance leader
provides technical support to the whole team. He or she is responsible to the maintenance administrator.
The co-leader is an assistant to the maintenance leader.
The user-liaison person is charged with linking the users and the maintenance team.
The maintenance administrator is the administrator with a range of responsibilities such as hiring, firing and promotion of staff.
The maintenance programmers perform problem diagnosis and implement change under the supervision of the maintenance leader.
It is important for it to have a mix of experienced and junior personnel.

Personnel Education and Training

Education and training in software maintenance is a traditionally neglected area.
The objectives and strategies of software maintenance education are discussed.

Objectives:

1. To raise the level of awareness

Maintenance personnel need to understand the processes and procedures of maintenance. In order to be able to do their job effectively. The reasons for this are as follows:

1. From a management perspective, maintenance managers must understand the specific needs of the maintenance environment in which they operate.
2. From a maintenance programmer's point of view, it is important for the programmer to recognise that maintenance is not just a peripheral activity in an organisation, it is at the heart of the organisation's functioning.
3. Despite recent improvements, software maintenance still has an image problem. One of the reasons for this is lack of understanding of what software maintenance really is; it is often perceived as simply effecting corrective change.
4. In circumstances where inexperienced staff (e.g., newly recruited graduates) are assigned to maintenance jobs, it is not uncommon for them to notice that the task they are working on is remote from what was taught to them in University or College courses.

2. To enhance Recognition:

In organisations whose operation depends on the successful evolution of their software systems, it needs to be recognised within the management structure that maintenance is a vital and valuable activity. When carried out effectively it can ensure successful running of systems and lead to increased customer satisfaction.

Education and Training Strategies:

1. University education: Many Universities currently run courses in software engineering, most of which touch upon software maintenance issues.

2. Conferences and workshops: Attending conferences and workshops nationally or internationally - e.g. those organised by bodies such as the IEEE Computer Society (http://www.computer.org/) and the Durham Research Institute in Software Evolution (http://www.dur.ac.uk/CSM/) - offer maintenance personnel the chance to meet others with similar experiences.

One of the advantages of such meetings is that the delegates can exchange ideas and identify areas for future collaborative work. One of the drawbacks is that due to the high cost, it may not possible for every member of a maintenance team to gain such exposure.

3. Hands on Experience: This remains the most valuable way to acquire the appropriate level of knowledge and skill required to undertake successful software maintenance.

Organizational Modes

There is a choice between combining development and maintenance activities or having a separate department. The decision to separate or combine the activities depends on factors such as the size of the organisation and the maintenance portfolio with which it must deal.

Combined Development and Maintenance:

1. Module Ownership:

The module ownership mode requires that each member of the team is assigned ownership of a module. The owner of a module is responsible for effecting any changes that need to be implemented in that module. The main advantage with this mode of organisation is that the module owner develops a high level of expertise in the module. Its weaknesses are:

1. Nobody is responsible for the overall software system.
2. The workload may not be evenly distributed.
3. It is difficult to implement enhancements due to unknown dependencies.
4. It is difficult to enforce coding standards.

2. Change Ownership:
Each person is responsible for one or more change no matter which modules are affected. That is, the person is also responsible for the analysis, specification, design, implementation, and testing of the change.

The strengths of the change ownership mode are:

a. There is a tendency to adhere to standards set for the whole software system.
b. Integrity of the change is ensured.
c. Changes can be coded and tested independently.
d. Code inspection tends to be taken seriously.

Its weaknesses are:

a. Training of new personnel takes much more time than it would for the module ownership mode. This is primarily because knowledge of the entire system is required.
b. Individuals do not have long-lasting responsibilities, but instead have a series of transient responsibilities.

3. Work Type:
The key feature of Work-Type mode is that there is 'departmentalisation' by work type; analysis, specification, etc. Those in the different departments work as a team but with clearly defined responsibilities and roles. The main strength of this arrangement is that members in each department develop specialised knowledge and skills. The drawback is the cost of co-ordinating the different departments.

4. Application Type:
Division is based on application areas such as health information systems or office automation. The advantage with this mode is that members of the team develop specialised application knowledge. Its drawback is the cost of co-ordinating of the various application domains.

Separate Maintenance Department

This mode of organisation requires a separate maintenance department. It is based on the need to maintain many system portfolios, and the increasing business need of always keeping software systems operational.

Its strengths are:

a. There is clear accountability.
b. It allows development staff to concentrate on development of new software systems.
c. It facilitates and motivates acceptance testing just after development.
d. It encourages high quality end-user service.

Its weaknesses are:

a. There is a danger of demotivation due to status differences.
b. The developers tend to lose system knowledge after the system is installed.
c. There is a high cost involved in the co-ordination of development and maintenance when needed.
d. There may be duplication of communication channels.

In the cases where there is a separate maintenance department, some organisations take measures to minimise the effect of the dichotomy by providing the maintenance team with support. This support can be provided by assigning some members of the development team to join the maintenance team on installation of the system.

These members were called as **maintenance escort.**

It is usually a temporary measure; the maintenance escorts return to development work after ensuring that the system functions according to the agreed specification. At times, the maintenance escorts may become permanent members of the maintenance team by virtue of their familiarity with the system during its development.

Chapter 11:

Configuration Management

Introduction

Configuration management is a means of keeping a hold of the process of software change, and of having confidence in the implementation of change. Configuration management looks at the overall system process and its constituent parts down to a certain level of detail.

The means by which the process of software development and evolution is controlled is called **configuration management.**

Configuration management in software maintenance differs from configuration management in software development because of the different environments in which the activities are carried out. Software maintenance is undertaken in the environment of a live system in use by a probably large user base.

Baseline - The arrangement of related entities that make up a particular software configuration. Any change made to a software system relates to a specific baseline. Baselines can be defined at each stage of the software life-cycle, for example functional baseline, design baseline, product baseline, etc.

Change - the act, process or result of being altered.

Software change control - keeping track of the process of making modifications to a software system.

Change control - keeping track of the process of making a modification.

Configuration - A mode of arrangement, confirmation, or outline; the relative position of the component parts of a system, for example the relative positions of the stars and planets in the solar system.

Software change control - keeping track of the process of making modifications to a software system.

Software configuration - The current state of the software system and the interrelationship between the constituent components. These would typically be the source code, the data files, and the documentation.

Software configuration management - Configuration management related specifically to software systems.

Software documentation - the written record of facts about a software system recorded with the intent to convey purpose, content, and clarity.

Variant - Source and object specialised to different platforms. For example, Microsoft Word for Windows for the PC and Microsoft Word for Windows for the Macintosh are variants of the same product.

Version - A version represents a small change to a software configuration. In software configuration management terms, one refers to versions of a given baseline rather than dealing with a proliferation of baselines.

Version control - Keeping track of baselines, versions of baselines and the relationships between them.

Configuration Management

The discipline of developing uniform descriptions of a complex product at discrete points in its life-cycle with a view to controlling systematically the way the product evolves.

Configuration management activities fall into **four** broad categories:

1. The identification of the components and changes

2. The control of the way the changes are made

3. Auditing the changes - making the current state visible so that adherence to requirements can be assessed

4. Status accounting - recording and documenting all the activities that have taken place.

In a large organisation, there might be a configuration management team with a configuration manager. In a small organisation, the duties of a configuration manager might be taken on by others along with other duties. But however, the task is organised, it is the same job that needs to be done.

All components of the system's configuration are recorded along with all relationships and dependencies between them. Any change - addition, deletion, modification - must be recorded and its effects upon the rest of the system's components 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. It is a management task both to assign these responsibilities and to conduct audits to see that they are carried out.

A major aim in configuration management and change control is reproducibility.

We might wish to reproduce an older version of a system if a new one has serious shortcomings or we might wish to reproduce specific functionality in a particular environment.

Objectives of configuration management:

Control:

The very essence of software maintenance is the evolutionary nature of software systems. If the process by which such systems evolve is not controlled, chaos can result. Configuration management is necessary because software systems have a long lifetime and are subject to change. Constant change to workable live software systems is bound to lead to problems without a proper means of control.

Consistency:

Configuration management is necessary to ensure the production of consistent sets of software products.

Consistent sets of documents

Software projects in their development will produce reams of documents. Some will be vital to the maintenance phase and these need to be identified.
However, many documents which are vital during the development process might not be needed subsequently.

It is very important to identify which these documents are, and decide what procedures are to be applied to keeping track of them. As well as standardised document identification which allows for versioning and indexing, standard reporting mechanisms should be put in place.

Consistent software releases

A large area of responsibility is the management of the release of changed systems into the world. Standardised version and product release control is needed to keep track of what is where.

Cost:

A major aim in configuration management is to ensure that the changes are made such that overall costs are minimised. This means cost in terms of future maintenance as well as the cost of the immediate change.

It is easy to see that without consistent and accurate records and documentation, it will be all but impossible to keep track.

With Proper Procedures, it will enable in identification of:

the software version in use by a particular user - attempting a repair of the wrong version will only make things worse;

the options for upgrade - it is necessary to know which options are viable in the user's environment;

the levels at which different versions are compatible - the structure of data files, for example, might change between versions or releases, and attempting to run an inappropriate version might lead to corruption of the user's data;

the source code corresponding to a specific version - if, as in the example, the only option is a new release of version 1.2 as version 1.2a, it is vital to be able to trace and reproduce the source code for version 1.2 despite the fact this might not correspond to any executable version currently in use.

Without a structured approach it is impossible to guarantee

i. the integrity of the system,

ii. that the course of evolution is traceable or

iii. that correlation between all the different parts of the system is identifiable.

Configuration management is more than just a set of methodologies and tools. An organisation will select configuration management methodologies appropriate to its specific needs but considerable skill is needed in both selection of the right techniques and methodologies and the effective imposition of them upon the software development and maintenance process.

Change Control

Change control concerns the specific area comprising the sequence of events starting with a request for a change to a live software system and ending with either the approval of a system incorporating the change or a rejection of the request for change.

Activities which comprise change control are:

- Selection from the top of a priority list.
- Reproduction of the problem (if there is one).
- Analysis of code (and specifications if available).
- Incorporation of change.
- Design of changes and tests.
- Quality assurance.

Change control ensures that changes to a system are made in a controlled way such that their effects can be predicted. Full change control procedures can be time-consuming and are not necessarily appropriate

at every stage of the evolution of a system. There are cases where the process of deciding and approving change can be done less formally.

Responsibility of management in control change:

Deciding if the change should be made:
A potential change must be costed. The cost of making the change must be balanced against the benefit that will accrue from it.

It is the job of the Change Control Board to decide whether to accept a request for change. It is usual to institute a change request form that gives details of the change requested and the action taken. Change request forms are a very useful form of documentation. The definition of the exact format of a change request form and the information contained within it is a job for the configuration management team although in some cases it will be necessary to conform to client standards.

Managing the implementation of the change:
The ramifications of making a change must be assessed. This assessment will have begun as part of the costing process.

Verifying the quality:
Implementation of a change should be subject to quality control. A new version of the system should not be released until it has satisfied a quality control process.

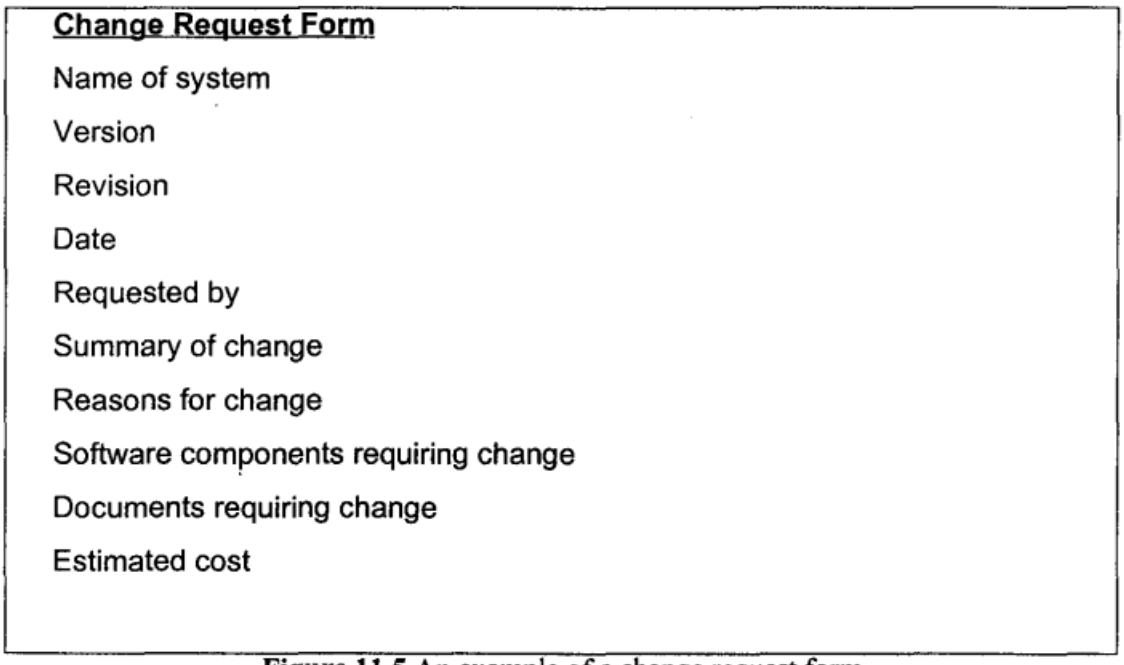

Figure 11.5 An example of a change request form

It is essential to be able to retain and manage the system components. In order to do this, accurate and up-to-date information about them must always be available. The means of making such information available is through documentation.

Documentation

Documentation is integral to the whole process of management and control. It has a major role to play in making processes and procedures visible and thereby allowing effective control. The recording process usually begins when the need for the system is conceived and continues until the system is no longer in use.

Categories of Documentation:

1. User Documentation

User documentation refers to those documents containing descriptions of the functions of a system without reference to how these functions are implemented.

2. System Documentation

System documentation contains documents which describe all facets of the system, including analysis, specification, design, implementation, testing, security, error diagnosis and recovery.

The user documentation and system documentation are further split into separate documents, each of which contains a description of some aspect of the software system

Table 7.1 Types and functions of documentation

Type	Constituent document	Function
User documentation	1. System overview	Provides general description of system functions
	2. Installation guide	Describes how to set up the system, customise it to local needs, and configure it to particular hardware and other software systems
	3. Beginner's guide / tutorial	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	Serves as a factual lookup
	7. System administration	Provides information on services such as networking, security and upgrading
System documentation	1. System rationale	Describes the objective of the entire system
	2. Requirements analysis / specification	Provides information on the exact requirements for the system as agreed between the user and the developer / maintainer
	3. Specification / design	Provides description: (i) of how the system requirements are implemented (ii) of 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
	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 descriptions of all terms that relate to the software system in question

There are three classes of documentation:

User Manual:

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.

Operator Manual:

The operator manual describes how to use the system as well as giving instructions on how to recover from faults.

Maintenance Manual:

The maintenance manual contains details of the functional specification, software design, high quality code listings, test data and results.

Although the list of software documents is not exhaustive, it contains many of the documents typically found in a system. The type, content and sometimes name of each document will vary between systems. This variation is caused by several factors that include:

Development methodology: The approach used to develop software systems differs from one organisation to another. The type of system documentation produced will depend on the approach used to develop the software.

Category of customer: The relation of an individual or organisation to a software system determines the subset of documents that accompany its copy of the system. For example, a software developer will sell user manuals and object code to software dealers. The same developer might sell system documentation as well as source code listings to a customer who modifies the system to build other products. As such, different customers may receive different documentation sets for the same software system.

Version of the system: System upgrades will be accompanied by additional documents such as enhancement booklets, which convey information on the new features of the system, and other literature on how to upgrade existing systems.

Role of Software Documentation:

To facilitate program comprehension:

The function of documentation in the understanding and subsequent modification of the software cannot be overemphasised. Prior to undertaking any software maintenance work, the maintainer should have access to as much information as possible about the whole system. Due to the high turnover of staff within the software industry, it may not always be possible to contact developers of the system for information about it.

As such, maintainers need to have access to documents about the system in order to enable them to understand the rationale, functionality, and other development issues. Each system document has a specific function

To Act as a guide to the user:

Documentation aimed at users of a system is usually the first contact they have with the system [255]. The user documentation that comes with a system is used for various purposes that include:

1. Providing an initial and accurate description of what the system can do. As such, the user can decide whether the system can satisfy his or her needs. In order to achieve this, the documents must be written and arranged in such a way that the user can easily find what is required.
2. Providing information that enables the user to install the system and customise it to local needs.
3. Providing technical information on how to handle malfunctions.

To complement the system:

Documentation forms an integral and essential part of the entire software system. Osborne argues that 'without the documentation, there is little assurance that the software satisfies stated requirements or that the organisation will be able to maintain it'.

Producing and Maintaining Documents

Osborne contends that the cost of maintaining a software system is proportional to the effectiveness of the documentation which describes what the system does as well as the logic used to accomplish its tasks.

It is not only what the documents contain those matters, but how the material is presented. Some authors have suggested several guidelines on producing software documents:

1. Writing style: adhering to guidelines for clear and understandable text, for example using the active rather than the passive mode, splitting the text into manageable chunks and repeating complex explanations in different ways.

2. Adhering to document standards: for example, standard cover sheets can ensure traceability of documents. Standard fonts, styles and numbering systems can make it easier for the reader to switch between documents - there will be no need to adapt to different styles for different documents.

3. Standards and quality assessment: putting documents through a quality assessment process will help to ensure conformance to standards.

4. Documentation techniques:

a) To ensure that documentation is up to date, procedures should be put in place to encourage the use of adequate resource for the updating of documents concurrent with system updates.

b) The use of good design methodologies, for example good design methods and practices such as structured programming, use of high-level control structures, meaningful identifier names and consistent style, reduces the need for low-level documentation. This is particularly helpful during maintenance.

5. Documentation support tools: support tools exist to help with the classification and updating of documentation. Appropriate tools can do much towards ensuring consistency in documents.

A common problem that confronts maintainers of software systems is ensuring consistency across all the documents when the software system changes. The usual solution to this problem is to keep a record alongside the documents of the relationships and dependencies of not only the documents, but also parts of the documents.

Software systems are modified and enhanced by individuals who were not involved with the initial development. In order to ensure that a system can be updated quickly and reliably, it is necessary to record all information relating to the evolution of the software. When the system is changed, all documents that are affected must be updated as and when the changes occur.

Chapter 13:

Building And Sustaining Maintainability

High level language - a computer programming language that is like a natural language and that requires each statement to be translated or interpreted into machine language prior to execution.
Impact analysis - the determination of the major effects of a proposed project or change.
Object oriented programming - computer programming in which code and data pertaining to a single entity (object) are encapsulated, and communicate with the rest of the system via messages.

Quality assurance - the systematic monitoring and evaluation of aspects of a project, service, or facility to ensure that necessary standards of excellence are being met.

Quality Assurance

Correctness and Maintainability as the two most important quality factors that a system can possess. The discussion here is of building maintainability into a system and, as such, maintainability will not be looked at as a quality factor by itself but rather as something which is affected by the other quality factors - fitness for purpose, correctness, portability, testability, usability, reliability, efficiency, integrity, reusability, and interoperability.

1. Fitness for Purpose:

Fitness for purpose - does the product do the job it was intended to do - is an obvious criterion by which to measure quality.

2. Correctness:

Building correctness into a system has the obvious advantage that less time will be spent on corrective maintenance. A maintenance-conscious life-cycle model will help maintain the correctness of a system.

3. Portability:

Portability encompasses many things, for example moves between

- hardware platforms, for example porting a system from a VAX to a PC;
- operating systems, for example from Windows to LINUX to widen a market, from VAX/VMS to Windows when an organisation moves from a VAX base to a PC base;
- programming languages, for example upgrading a system from DOS-based to Windows-based by rewriting in a different language or rewriting a system to adhere to a new in-house standard, for example Fortran 77 to ANSI C**.**
- countries, for example an organisation may wish to widen its market for a product from the UK to Europe. The product would need to be available in all European languages.

Building for portability will enhance maintainability by easing moves between platforms, languages and so on. The building of portability into a system means the avoidance of features which tie a software system to a particular platform or language. Adhering to standards can help portability as can appropriate decisions regarding what should be hard-coded and what should not.

4. Testability:

A system that is easy to test is also easier to change effectively because it is easier to test the changes made. It does not automatically follow that a system that is hard to test will be harder to change per se but if the system cannot be tested effectively, it is hard to engender confidence that modifications have been carried out successfully.

5. Usability:
If a system for any reason is not used, it may as well not exist. Maintenance is only an issue for a system that is used and that evolves with use.

6. Reliability:
Varying degrees of importance can be attached to reliability, depending upon the application. Building in a high degree of reliability can be costly and is not always necessary.

7. Efficiency:
The efficiency of a system, how it makes use of a computer's resources, is not always easy to quantify.

8. Integrity:
The integrity of a system can be interpreted in two ways:

1. Is the system safe from unauthorised access?
2. Can system integrity be guaranteed from a configuration management point of view? Has the system been built from a consistent and reproducible set of modules?

9. Reusability:
Reusability is a factor that is central to the maintainability of a system.

10. Interoperability:
The ability of a system to interact with other systems is a very important factor.

Fourth Generation Languages (4GLs)

First-generation Language:
The use of first-generation languages entails programming computers using binary notation. These languages require a good understanding of low-level details of the machine such as physical storage locations and registers. Programs written using these languages are very dependent on the specific machine for which they are written. An example is machine code.

Second-Generation Language:
Second-generation languages were an improvement of the firstgeneration ones. Instead of specifying the physical location in the computer, symbolic addresses are used to designate locations in memory. Second-generation languages are slightly less machine dependent. An example is symbolic assembler.

Third Generation Languages:
AKA high-level languages, are more independent of the machine than second-generation languages and as such their use does not require knowledge of the machine instruction set. Thus, programs can be easily ported to different machines. Also, these languages are generally standardised and procedural in nature - their use requires adherence to some rules and regulations. These languages are often used to solve scientific or commercial problems. Examples are Pascal, Cobol, Ada and Modula-2.

They still present several difficulties which include:

- Writing and debugging programs is a significantly slow, difficult, and expensive process. Often these difficulties contribute to late delivery of software products.
- Many of them can only be used effectively by professional programmers. If there is a dearth of the relevant professionals, projects will be significantly slowed.

- The implementation of changes to complex software systems is slow, difficult-and hence can greatly increase maintenance costs.
- Several lines of code need to be written to solve a relatively small problem, thereby impeding programmer productivity.

In response to the above problems, 4GLs were created. In other words, the advent of 4GLs was seen as a way of reducing dependence on professional programmers and giving users the power to obtain fast results to support their day-to-day operation without the need for extensive programming.
It consists of simple query languages, complex query and update languages, report generators, graphics languages, decision-support languages, application generators, specification languages, very-high-level programming languages, parameterised application packages and application languages.

Properties of 4GLs

- easy to use.
- can be employed by non-professional programmers to obtain results.
- They use a database management system directly.
- They require an order of magnitude fewer instructions than other conventional languages such as Cobol.
- Where possible, they use non-procedural code.
- Where possible, intelligent default assumptions about what the user wants are made.
- They are designed for on-line operations.
- They encourage or enforce structured code. This has proved true only to a point for early 4GLs. Maintenance programmers struggling with badly written 4GL code can testify to reams of ill-structured code. However, in relation to early alternatives, the statement holds.
- They make it easy to understand someone else's code. Once again, this characteristic and the next one were of greater significance when compared to 1980's alternatives.
- They are designed for easy debugging.
- Non-technical users can learn to use a subset of the language after a two-day training course.
- They allow results to be obtained in an order of magnitude less time than with Cobol or PLII. It is not clear that this characteristic is backed up by rigorous empirical study.
- Where possible, documentation is automated.

4GLs Impact on maintenance

1. Increased Productivity

One of the strengths of 4GLs is that they enable more rapid implementation of change, thereby increasing productivity.

2. Reduction in Cost

Due to the reduction in time required to develop and maintain applications using 4GLs, they tend to cost less than conventional application development.

3. Ease of understanding

shorten comprehension time

4. Automatic Documentation

A large proportion of the documentation is generated automatically.

5. Reduction in Workload

The user-friendly and easy to learn nature of many 4GLs allowed end-users to implement modifications with little or no assistance, thereby reducing the workload for the maintenance personnel.

Weakness of 4GLs

1. Application Specific

The very fact that many of these languages are targeted at specific application domains makes it difficult to use them for applications other than those for which they were originally designed.

2. Proprietary

Many 4GLs are not compatible; that is, they are proprietary and are not 'open languages'.

3. Hyped Ease of Use

There have been a lot of claims in the literature that 4GLs are designed for non-professional programmers.

4. Poor Design

The ease-of-use issue means that systems can be developed by people with no expertise or grounding in software engineering, or by people with no thorough understanding of the business problem being solved.

Object Oriented Paradigms

The object-oriented paradigm was developed to handle the design, development, and maintenance of industrial-strength software. The object-oriented paradigm was developed to handle the design, development, and maintenance of industrial-strength software. Such systems, e.g., air traffic control systems, airline ticket reservation systems and spacecraft control systems, are large and dynamic networks of interacting systems.

Impact on Maintenance

One of its main advantages is that there is a common view of what happens in the real world and how this is implemented in the software, and as such, the transformation from the analysis phase to the implementation phase is much clearer and less error prone.

More advantages are:

- It yields smaller systems due to the reuse of common mechanisms.
- It facilitates the production of systems that are resilient to change and hence easily evolvable considering that its design is based on a stable intermediate form. The potential of object orientation to increase maintainability is a contributory factor to its popularity.
- The risks associated with complex systems are reduced because object-oriented systems are built to evolve incrementally from smaller components that have been shown to satisfy the requirements.
- Products from object-oriented decomposition can be stored in libraries and reused.

Disadvantages:

- Their application is not universal.
- It takes time to bring a new paradigm on stream.
- There is a large body of code in existence requiring maintenance. Thus, even if object-orientation becomes the accepted paradigm for software development, the need for expertise in non-object-oriented techniques will persist for a very long time.
- It is not necessarily appropriate or cost effective to convert existing systems.

Approaches to object-oriented languages:

The first is to rewrite the whole system. In other words, throw away the current system and start from scratch, but develop the new system from an object-oriented perspective. This may be an option for small systems of a few thousand lines of code. For many large systems - usually hundreds of thousands or several million lines of code - that represent a significant part of the assets of the organisations that own them, a better approach is required.

The second approach to migration, especially appropriate in the early days of moves to object-oriented systems, was to use object-oriented analysis as a 'springboard'; that is, perform object-oriented analysis of the existing system and implement it using a mainstream but non-object-oriented language such as Cobol or C and then migrate to a suitable object-oriented language at some later time. The key advantage of this approach was that it avoided the risk of choosing an object-oriented programming language that would prove a bad choice as object-oriented languages themselves evolved.
The key advantage of this approach was that it avoided the risk of choosing an object-oriented programming language that would prove a bad choice as object-oriented languages themselves evolved.

The third, and possibly the approach preferred by most organisations, is that which permits organisations to reap the benefits that object orientation offers as well as securing the investment that has been made in their existing systems. This objective can be achieved through techniques such as abstraction engineering and object wrapping.

The process of abstraction engineering involves taking the existing non-object system and modelling it as a set of objects and operations. This enables the identification of key reusable components which themselves can be assembled to form new objects that can be used for both perfective and preventive maintenance of the existing system as well as the development of new systems.

After identifying objects through abstraction engineering, programs which allow the objects to be accessed in a formal and controlled fashion are developed through a process known as object wrapping. These programs are called wrappers. The wrappers serve as interfaces to these objects and are what any prospective user of the object will be presented with. The wrapped objects can then be implemented in one or more object-oriented programming languages. The main advantage of object wrapping is that after an object has been wrapped, it can be reused in several other systems as if it was developed from scratch.

Retraining Personnel

The key issues that preoccupy the software community in relation to this change process are usually technical and managerial in nature. An area that has been neglected is the retraining of personnel - analysts, designers, and programmers - already involved in mainstream technologies or programming languages, such as structured programming and Cobol. Such retraining is essential since it can be used to capitalise on the existing knowledge base that the personnel have acquired by virtue of their experience.

Undoubtedly the retraining process is not only an expensive activity but also a difficult task.

Although little attention has been paid to the retraining of personnel in connection with the migration of legacy systems, there are examples in which these systems have been reengineered successfully using object-oriented techniques.

Chapter 14:

Maintenance Tools

Tool - implement or device used to carry out functions automatically or manually.

Software maintenance tool - an artefact used to carry out automatically a function relevant to software change.

Criteria for selecting tools:

There are several factors that should be taken into consideration:

Capability:

The tool must be capable of supporting the task to be performed. When a technique or method is to be supported by a tool, it is necessary to ensure first that it works without a tool; that is, by hand.

Features:

Particular features may be required of a tool. The importance of each of these features should be rated and the tool selected accordingly. As a simple example, a useful word processor will need to provide not just an editor, but also other features such as a spelling checker, thesaurus, drawing and search facilities.

Cost and Benefits:

The cost of introducing a tool needs to be weighed against the benefits. The benefits that the tool brings need to be evaluated in terms of indicators such as product quality, productivity, responsiveness, cost reduction, and extent of overlap or dichotomy between different groups with respect to their way of doing things.

Platform:

The platform refers to the specific hardware and software environments on which the tool runs.

Programming language:

This refers to the language that will be used to write the source code. To be on the safe side, it is important to obtain a tool that supports a language that is already (or is likely to become) an industry standard. This is particularly important in situations where there is migration to a new paradigm, for example migration to object-oriented development.

Ease of use:

The ease with which users can get to grips with the tool determines, to some extent, its acceptability. Usually, a tool that has a similar 'feel' to the tools that users are already familiar with tends to be accepted more easily than one which is radically different.

Openness of architecture:

This is particularly important in situations where the desired tool needs to run in conjunction with existing tools. Another reason for selecting a tool with an open architecture is that in very complex maintenance problems a single product from one vendor may not be capable of performing all the required tasks.

Stability of vendor:

it is essential to investigate the background of any company being considered as a supplier of a tool. If the tool is one with an open architecture, then this factor may not be so important.

Organisational culture:

Organisations usually have a particular way in which they operate; a working culture and work patterns. In order to increase the chances of the tool being accepted by the target users, it is essential to take such culture and work patterns into consideration.

Taxonomy of Tools:

The categories of tasks for which tools will be discussed are:

1. Program understanding and reverse engineering
2. Testing
3. Configuration management
4. Documentation and measurement.

Tools For Comprehension and Reverse engineering:

1. Program Slicer:

One of the major problems with software maintenance is coping. With the size of the program source code. It is important that a programmer can select and view only those parts of the program that are affected by a proposed change without being distracted by the irrelevant parts. One technique that helps with this problem is known as slicing - a mechanical process of marking all sections of a program text that may influence the value of a variable at a given point in the program. The tool used to support slicing is known as a program slicer. The program slicer also displays data links and related characteristics to enable the programmer to track the effect of changes.

2. Static Analyzer:

To understand a program, there is usually a need to obtain information about different aspects of the program such as modules, procedures, variables, data elements, objects and classes, and class hierarchy. A static analyser allows derivation of this information through careful and deep examination of the program text.

Generally, a static analyser:

- allows general viewing of the program text - serves as a browser;
- generates summaries of contents and usage of selected elements in the program text such as variables or objects

3. Dynamic Analyzer:

When studying a software system with the aim of changing it, simply examining the program text - static analysis - may not provide all the necessary information. Thus, there is a need to control and analyse various aspects of the program when it is executing. A tool that can be used to support this process is known as a dynamic analyser.

Generally, the dynamic analyser

- allows a maintainer to trace the execution path of the system while it is running - it acts as a tracer. This permits the maintainer to determine the paths that will be affected by a change and those through which a change must be made.

4. Data Flow Analyzer:

A data flow analyser is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program and to backtrack. This is particularly important when there is a need for impact analysis: studying the effect of a change on other parts of the system. By tracking the flow of data and control, the maintainer can obtain information such as where a variable obtained its value and which parts of the program are affected by the modification of the variable.

Generally, a data flow analyser also:

- allows analysis of program text to promote understanding of the underlying logic of the program;
- assists in showing the relationship between the different components of the system;
- provides pretty-printers that allow the user to select and display different views of the system.

5. Cross Referencer:

The cross-reference is a tool that generates an index of the usage of a given program entity. For example, it can produce information on the declarations of a variable and all the sections in the program in which it has been set and used. During the implementation of a change, the information this tool generates helps the maintainer to focus and localise attention on those parts of the program that are affected by the change.

6. Dependency Analyzer:

A dependency analyser helps the maintainer to analyse and understand the interrelationships between entities in a program. This tool is particularly useful in situations where logically related entities, such as variables, may be physically far apart in the program.

Generally, a dependency analyser:

- can provide capabilities that allow a maintainer to set up and query a database of the dependencies in a program. Information on dependencies can also be used to determine the effect of a change and to identify redundant relationships between entities;
- provides a graphical representation of the dependencies in a program where the node in the graph represents a program entity and an arc represents the dependency between entities.

7. Transformation Tool:

A transformation tool converts programs between different forms of representations, usually between text and graphics; for example, transforming code to visual form and vice versa. Because of the impact that visual representations can have on comprehension, the use of a transformation tool can help the maintainer view and understand the system in a way that would not be possible with, for example, just the textual representation. The tool usually comes with a browser and editor, which are used to edit the program in any of its representations.