Chapter 1:

Introduction to the Basic Concepts

Software Maintenance:

modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Components of a Software System/what does a software made up of:

Table 1.1 Components of a software system

Software Components	Examples		
Program	1 Source code 2 Object code		
Documentation	1 Analysis / specification:	(a) (b) (c)	Formal specification Context diagram Data flow diagrams
	2 Design:	(a) (b)	Flowcharts Entity-relationship charts
	3 Implementation:	(a) (b)	Source code listings Cross-reference listings
	4 Testing:	(a) (b)	Test data Test results
Operating procedures	1 Instructions to set up and use the software system 2 Instructions on how to react to system failures		

Different views of software maintenance:

- the bug-fixing view - maintenance is the detection and correction of errors,
- the need-to-adapt view - maintenance is making changes to software when its operational environment or original requirement changes,
- the user-support view - maintenance is the provision of support to users.

How New Development and Maintenance Activities Differ:

New development is, within certain constraints, done on a green field site. Maintenance must work within the parameters and constraints of an existing system.

To explain the difference between new development and software maintenance, An interesting analogy is, the addition of functional requirements to a live system, to the addition of a new room to an existing building:

"The architect and the builders must take care not to weaken the existing structure when additions are made. Although the costs of the new room usually will be lower than the costs of constructing an entirely new building, the costs per square foot may be much higher because of the need to remove existing walls, reroute plumbing and electrical circuits and take special care to avoid disrupting the current site"

Why software maintenance is needed?

To provide continuity of service:

- Systems need to keep running. There can be severe consequences to system failure such as serious inconvenience or significant financial implications. Maintenance activities aimed at keeping a system operational include bug-fixing, recovering from failure, and accommodating changes in the operating system and hardware.

To support mandatory upgrades:

- This type of change would be necessary because of such things as amendments to government regulations e.g., changes in tax laws will necessitate modifications in the software used by tax offices. Additionally, the need to maintain a competitive edge over rival products will trigger this kind of change.

To support user requests for improvements:

- Overall, the better a system is, the more it will be used and the more the users will request enhancements in functionality. There may also be requirements for better performance and customisation to local working patterns.

To facilitate future maintenance work:

- It does not take long to learn that shortcuts at the software development stage are very costly in the long run. It is often financially and commercially justifiable to initiate change solely to make future maintenance easier.

If a system is used, it is never finished because it will always need to evolve to meet the requirements of the changing world in which it operates.

Maintaining Systems Effectively

In order to maintain systems effectively, a good grounding in the relevant theory is essential and certain skills must be learnt. Software maintenance is a key discipline, because it is the means by which systems remain operational and cope efficiently in a world ever more reliant on software systems. Maintenance activities are far-reaching. The maintenance practitioner needs to understand the past and appreciate future impact. As a maintenance engineer, you need to know whether you are maintaining a system that must operate for the next five minutes or the next five decades.

Categorising Software Change

Software change may be needed, and initiated, for several reasons and can be categorised accordingly.

- Modification initiated by defects in the software.
- Change driven by the need to accommodate modifications in the environment of the software system.
- Change undertaken to expand the existing requirements of a system.
- Change undertaken to prevent malfunctions.

The categorising of software change is not simply an academic exercise. It is vital in understanding when and how to make changes, how to assign resources and how to prioritise requests for change

Chapter 2:

The Maintenance Framework

Software Maintenance Framework

To a large extent the requirement for software systems to evolve in order to accommodate changing user needs contributes to the high maintenance costs. Additionally, there are other factors which contribute indirectly by hindering. A Software Maintenance Framework will be used to discuss some of these factors.

Component	Feature
1. User requirements	▪ Requests for additional functionality, error correction and improving maintainability ▪ Request for non-programming-related support
2. Organisational environment	▪ Change in policies ▪ Competition in the market place
3. Operational environment	▪ Hardware innovations ▪ Software innovations
4. Maintenance process	▪ Capturing requirements ▪ Creativity and undocumented assumptions ▪ Variation in programming practice ▪ Paradigm shift ▪ 'Dead' paradigms for 'living' systems ▪ Error detection and correction
5. Software product	▪ Maturity and difficulty of application domain ▪ Quality of documentation ▪ Malleability of programs ▪ Complexity of programs ▪ Program structure ▪ Inherent quality
6. Maintenance personnel	▪ Staff turnover ▪ Domain expertise

User:

The user in this context refers to individuals who use the system, regardless of their involvement in its development or maintenance.

Environment:

The environments affecting software systems are the operating environment and the organisational environment.

Operating Environment:

> Hardware innovations: The hardware platform on which a software system runs may be subject to change during the lifetime of the software. Such a change tends to affect the software in several ways. For example, when a processor is upgraded, compilers that previously produced machine code for that processor may need to be modified.

Software innovations: Like hardware, changes in the host software may warrant a corresponding modification in the software product. Operating systems, database management systems and compilers are examples of host software systems whose modification may affect other software products.

Organizational Environment:

Examples of the entities of the organisational environment are policies and imposed factors of business and taxation, and competition in the market place.

Change in policies: Many information systems have business rules and taxation policies incorporated into their program code. A business rule refers to the procedures used by an organisation in its day-to-day operation. A change in the business rule or taxation policy leads to a corresponding modification of the programs affected. For example, changes to Value Added Tax (VAT) rules necessitate modification of programs that use VAT rules in their computations.

Competition in the market place: Organisations producing similar software products are usually in competition. From a commercial point of view, organisations strive towards having a competitive edge over their rivals (by securing a significant proportion of the market for that product). This can imply carrying out substantial modifications to maintain the status of the product - reflected in the level of customer satisfaction - or to increase the existing 'client base'.

The user may not be billed directly for maintenance costs arising from changes in the organisational environment motivated by the need to keep a competitive edge. However, despite no direct financial input from the user, resources (both machine and human) still need to be allocated.

Maintenance Process:

The maintenance process itself is a major player in the software maintenance framework.

Capturing change requirements: This is the process of finding out exactly what changes are required. It poses a lot of problems. Firstly, it is fundamentally difficult to capture all requirements a priori. Requirements and user problems only really become clear when a system is in use. Many users know what they want but lack the ability to express it in a form understandable to the analyst or programmer. This is due to the 'information gap' defined earlier.

Variation in programming practice: This refers to differences in approach used for writing and maintaining programs. It involves the use of features or operations which impose a particular program structure. This tends to vary between individuals and organisations and may present difficulties if there is no consistency. Basic guidelines on good programming practice have been available for decades and aim to minimise future difficulties. Traditional guidelines would include: avoiding the use of 'GOTOs', the use of meaningful identifier names, logical program layout, and use of program commentary to document design and implementation rationale. There exist psychological arguments and empirical evidence, albeit sparse, which suggest that these features impact on program comprehension and hence can influence the amount of time required for effecting change.

'Dead' paradigms for 'living' systems: Many 'living systems' are developed using 'dead paradigms', that is, using the Fixed-Point Theorem of Information Systems [op. cit.]. Based on this theorem, there exists some point in time when everyone involved in the system thinks they know what they want and agree with everyone else. The resulting system is satisfactory only at the point at which it is delivered to the user. Thereafter, it becomes difficult - with few exceptions - to accommodate the changing needs of the users and their organisations. The extra flexibility to evolve that is provided

by interoperability goes some way towards alleviating this problem, but does not solve it. However, it should be noted that for a system to be built at all, it is essential that requirements are agreed - this is not the same as agreeing what the "finished" product is.

Error detection and correction: 'Error-free' software is non-existent. Software products have 'residual' errors which are difficult to detect even with the most powerful testing techniques and tools. The later these errors are discovered during the life-cycle of a software product, the more expensive they are to correct. The cost gets even higher if the errors are detected during the maintenance phase.

Paradigm shift: This refers to an alteration in the way we develop and maintain software [34]. Despite the enormous strides made in structure and reliability of programming languages, there still exists many systems developed using inadequate software tools. As well as the many systems still in use that were developed using low-level programming languages [211] and those developed prior to the advent of structured programming techniques, there are many programs in operation and in need of maintenance that were developed without the means to take advantage of the more advanced and more recently developed techniques. Such programs inherit a number of characteristics:

1. They were designed using techniques and methods that fail to communicate essential features such as program structure, data abstractions and function abstractions.
2. The programming languages and techniques used to write the code did not make visible and obvious the program structure, program interfaces, data structures and types, and functions of the system.
3. The constraints that affected their design no longer present a problem today.
4. The code can sometimes be riddled with non-standard or unorthodox constructs that make the programs difficult to understand, the classic example being the use of GOTOs.

In order to address these weaknesses and reap the benefits of modern development practices and the latest programming languages, existing programs may be restructured or completely rewritten. Examples of techniques and tools that can be used for this include: structured programming, object orientation, hierarchical program decomposition, reformatters and pretty-printers, automated code upgrading. It must be noted however that even programs developed using state of the art programming methods gradually lose their structure after being subjected to a series of unplanned and ad hoc 'quick fixes'. This continues until preventive maintenance is carried out to restore order to their structure.

Software Product

Aspects of a software product that contribute to the maintenance challenge include:

Maturity and difficulty of the application domain: The requirements of applications that have been widely used and well understood are less likely to undergo substantial modification on installation than those that are still in their infancy.

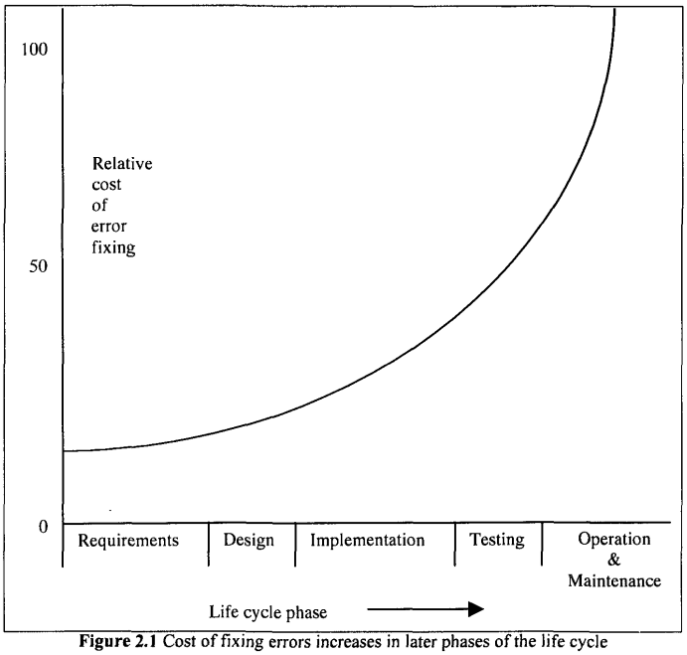


Figure 2.1 Cost of fixing errors increases in later phases of the life cycle

Quality of the documentation: The lack of up-to-date systems' documentation is one of the major problems that software maintainers face. Programs are often modified without a corresponding update of the documents affected. Even in environments where there are automatic documentation support tools, their contents may be inaccurate. Worse still, there may be no documentation at all. Inadequate documentation adversely affects maintenance productivity even for a programmer maintaining his / her own program, and in most cases, people are maintaining programs written by others.

Malleability of the programs: The malleable or 'soft' nature of software products makes them more vulnerable to undesirable modification than hardware items. With other more orthodox engineering deliverables, there is a well-defined approach to implementing a change.

Inherent quality: The nature of the evolution of a software product is very closely tied to the nature of its associated programs.

"A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version. "

This tendency for the system to decay as more changes are undertaken implies that preventive maintenance needs to be undertaken to restore order in the programs, thereby changing the product to a better and more sophisticated state.

Maintenance Personnel: Maintenance personnel include maintenance managers, analysts, designers, programmers, and testers. The personnel aspects that affect maintenance activities include the following:

- **Staff Turnover:** Due to the high staff turnover within the Information Technology industry, especially about software maintenance, most systems end up being maintained by people who are not the original authors.
- **Domain expertise:** The migration of staff to other projects or departments can mean that they end up working on a system for which they have neither the system domain knowledge nor the application domain knowledge. The lack of such knowledge may mean that the programmers can introduce changes to programs without being aware of their effects on other parts of the system - the ripple effect. This problem will be worsened by the absence of documentation. Even where documentation exists, it may be out of date or inadequate. These problems all translate to a huge maintenance expenditure.

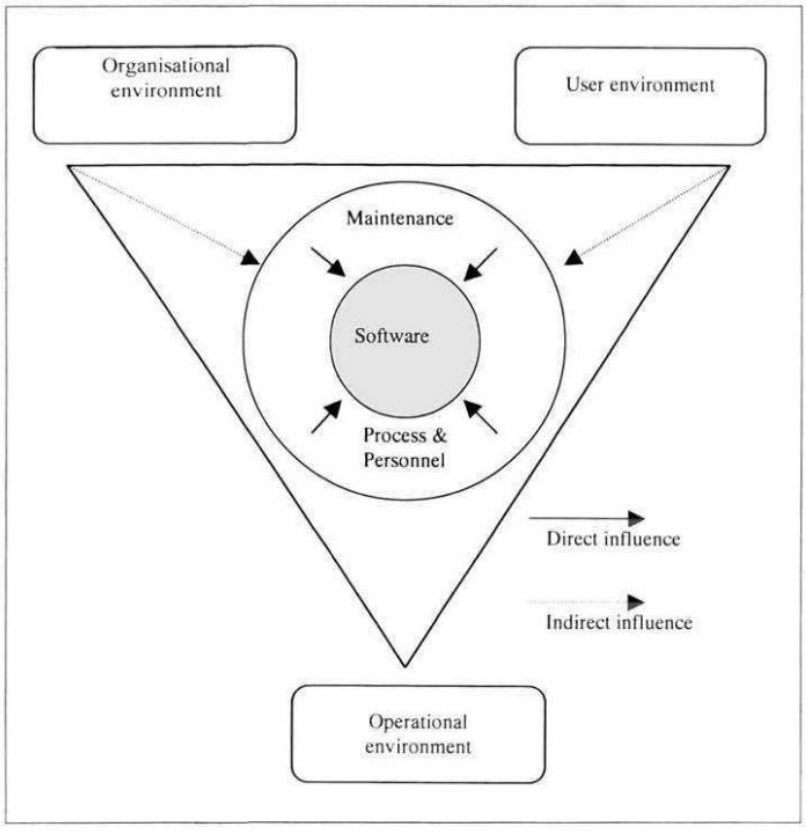
- **Working Practices:** Software systems do not change unless they are changed by people. The way the change is carried out is an important factor in how easy the resulting system will be to understand. Factors that can make the job more difficult include such things as:
  - a maintainer's desire to be creative (or 'clever');
  - the use of undocumented assumption sets;
  - undocumented design and implementation decisions. It should always be borne in mind that after time has elapsed, programmers find it difficult to understand their own code.

Relations Between the Maintenance Factors:

Three major types of relation and interaction that can be identified are product/environment, product/user and product/maintenance personnel.

Relation between product and environment: A software product does not exist in a vacuum, rather it can be seen as an entity which is hosted by its organisational and operational environments. As such, it inherits changes in the elements of these environments - taxation policies, software innovations, etc.

Relation between product and user: One of the objectives of a software product is to serve the needs of its users. The needs of the users change all the time. For the system to stay useful and acceptable it has to change to accommodate these changing requirements.



Interaction between personnel and product: The maintenance personnel who implement changes are themselves the conduit through which change is implemented. Changes in factors such as user requirements, the maintenance process, or the organisational and operational environments, will bring about the need for a change in the software product. However, the software product will not be affected until the maintenance personnel implement the changes. The type of maintenance process used and the nature of the maintenance personnel themselves, will affect the quality of the change.

Chapter 3:
Fundamentals of Software Change

Software Change:

Corrective Change: Corrective change refers to modification initiated by defects in the software. A defect can result from design errors, logic errors and coding errors.

> **Design errors** occur when, for example, 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 testing of 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.

All these errors, sometimes called 'residual errors' or 'bugs', prevent the software from conforming to its agreed specification.

Adaptive Change:

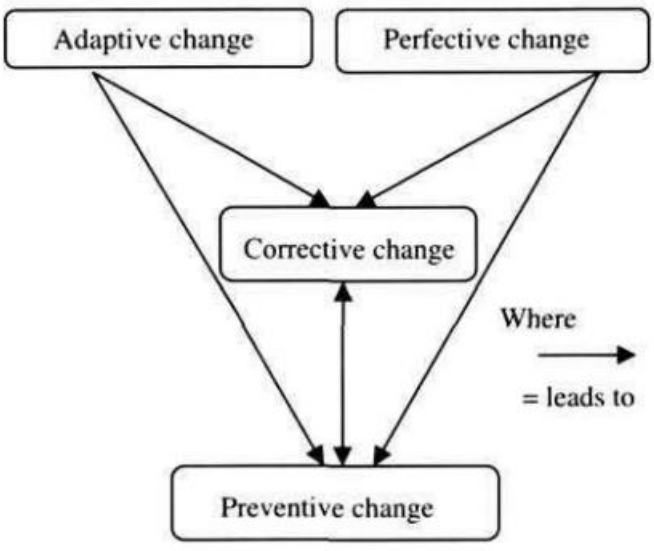
Adaptive change is a change driven by the need to accommodate modifications in the environment of the software system. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the system.

Perfective Change:

This term is used to describe changes undertaken to expand the existing requirements of a system. A successful piece of software tends to be subjected to a succession of changes resulting in an increase in its requirements. This is based on the premise that as the software becomes useful, the users experiment with new cases beyond the scope for which it was initially developed.

Preventive Change:

Preventive change is undertaken to prevent malfunctions or to improve maintainability of the software. The change is usually initiated from within the maintenance organisation with the intention of making programs easier to understand and hence making future maintenance work easier. Preventive change does not usually give rise to a substantial increase in the baseline functionality.



In practice, however, they are usually intertwined. For example, while modifying a program due to the introduction of a new operating system (adaptive change), obscure bugs may be introduced. The bugs must be traced and dealt with (corrective maintenance). Similarly, the introduction of a more efficient sorting algorithm into a data processing package (perfective maintenance) may require that the existing program code be restructured (preventive maintenance).

Incremental Release:

The changes made to a software product are not always done all together. The changes take place incrementally, with minor changes usually implemented while a system is in operation. Major enhancements are usually planned and incorporated, together with other minor changes, in a new release or upgrade.

Ongoing Support:

This category of maintenance work refers to the service provided to satisfy non-programming-related work requests. Ongoing support, although not a change, is essential for successful communication of desired changes. The objectives of ongoing support include effective communication between maintenance and end-user personnel, training of end-users and providing business information to users and their organisations to aid decision making.

Effective communication: This is essential between the parties affected by changes to a software system. Maintenance is the most customer-intensive part of the software life-cycle since a greater proportion of maintenance effort is spent providing enhancements requested by customers than is spent on other types of system change. The time and resources spent need to be justified by customer satisfaction.

Training of end-users: Training refers to the process of equipping users with sufficient knowledge and skills to enable them use a system to its full potential. This can be achieved in different ways. Users are traditionally supplied with supposedly comprehensive and easy-to-use manuals, which are designed to facilitate the use of the system. Manuals, however, do not always provide the information needed or do not provide the information in an easily understood way. Users, as a result, resort to other means such as on-line help or telephone queries. If this fails, arrangements can be made for on-site visits. Formal or informal short courses are arranged to train users in the use of the software and how to deal with problems that arise. In either case, the degree of commitment depends on the maintenance agreement between both parties.

Providing business information: Users need various types of timely and accurate information to enable them take strategic business decisions. For instance, a company planning to make major enhancements to its database management system may first want to find out the cost of such an operation. With such information, the company is in a better position to know whether it is more economical to enhance the existing system or to replace it completely.

Chapter 4:
Limitations and Economic Implications to Software Change

Maintenance crisis

The current predicament where the demand for high quality sophisticated software systems far outstrips the supply of such systems, but existing systems lack the capacity to evolve appropriately with technological advancement.

Potential Solutions to Maintenance Problems

Budget and Effort Reallocation: Based on the observation that software maintenance costs at least as much as new development, some authors have proposed that rather than allocating less resource to develop unmaintainable or difficult-to maintain systems, more time and resource should be invested in the development - specification and design - of more maintainable systems. The use of more advanced requirement specification approaches, design techniques and tools, quality assurance procedures and standards such as the ISO 9000 series and maintenance standards are aimed at addressing this issue. It is believed that the deployment of these techniques, tools and standards earlier on in the development life-cycle will lead to more maintainable systems.

Complete Replacement of the System: After having examined the problems that the maintenance of legacy systems poses for many medium-size to large organizations, particularly their unwelcome economic impact on the software budgets of these organizations, one might be tempted to suggest that if maintaining an existing system costs as much as developing a new one, why not develop a new system from scratch? This suggestion is understandable, but in practice it is not so simple. The risks and costs associated with complete system replacement are very high and the following must be considered. i.e.

- Economic constraints
- Residual errors in new systems
- Database of information

Maintenance of the Existing System: Complete replacement of the system is not usually a viable option. An operational system can be an asset to an organization in terms of the investment in technical knowledge and the working culture engendered. The 'budget/effort reallocation' approach, though theoretically convincing, is somewhat difficult to pursue. Even if the latter were possible, it would be too late for systems that are already in what is being termed the maintenance crisis. As an alternative, the current system needs to have the potential to evolve to a higher state, providing more sophisticated user-driven functionality, the capability of deploying cutting edge technology, and of allowing the integration of other systems in a cost-effective manner. Several techniques, methods, tools and management practices are used to meet these goals. These issues provide the underlying framework for much of the remainder of this book.

Limitations to Software Change:

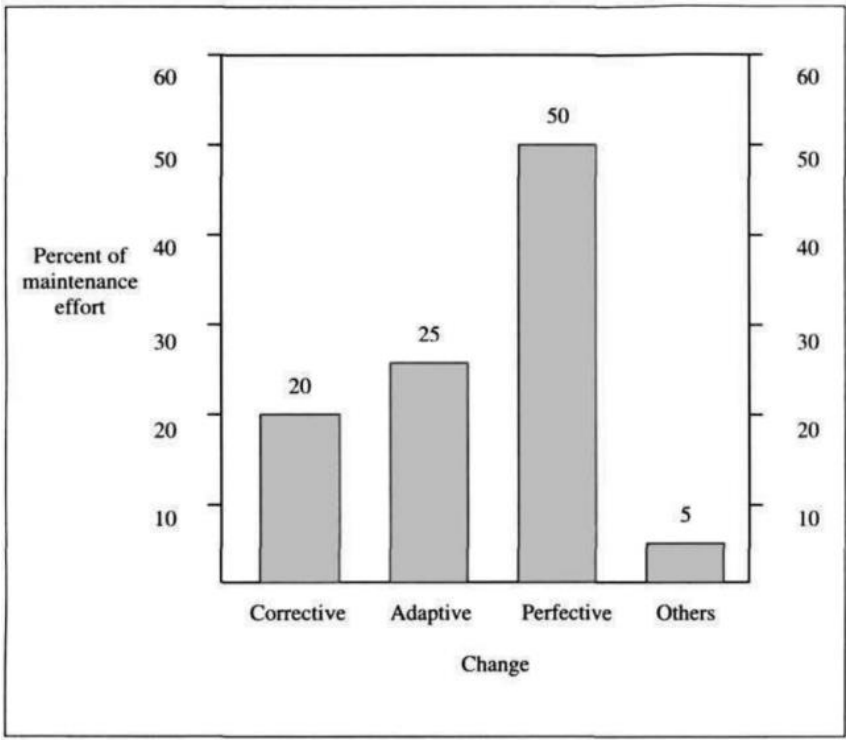
Resource Limitations: One of the major impediments to the quality and productivity of maintenance activities is the lack of resources. The lack of skilled and trained maintenance programmers, the lack of suitable tools and environment to support their work, and lack of sufficient budget allocation are issues that hold back change.

Quality of existing systems: In some 'old' systems, quality can be so poor that any change can lead to unpredictable ripple effects and a potential collapse of the system. Industry spends significant sums of money on maintaining obsolete code, even to the extent of maintaining obsolete hardware, because the risk of unforeseen ripple effects of excising the code is too great. Quality can become so poor that change is virtually impossible. This means that errors cannot be addressed and the system cannot evolve with progress in technology. This can lead to users tied into old systems (software and hardware), with huge maintenance costs, and with a system that provides less functionality that cheaper competitor systems. They cannot swap because of the data tied into the old system.

Organizational Strategy: The desire to be on a par with other organizations, especially rivals, can be a great determinant of the size of a maintenance budget. It is often these strategic decisions, far more than objective analysis of the problem being tackled that determines the maintenance budget. This is not the ideal way to ensure the job gets done in the best way.

Inertia: The resistance to change by users may prevent modification to a software product, however important or potentially profitable such change may be. This is not just unthinking stubbornness. For many decades, users have been promised the earth from software enterprises who can see the potential of the technology, but who have been unable to deliver for a variety of reasons. Sometimes the hardware has been inadequate (the days of overheating valves spiked many a budding software engineer's guns). Sometimes the theoretical base has been non-existent, and software developers have been building on quicksand without realizing it.

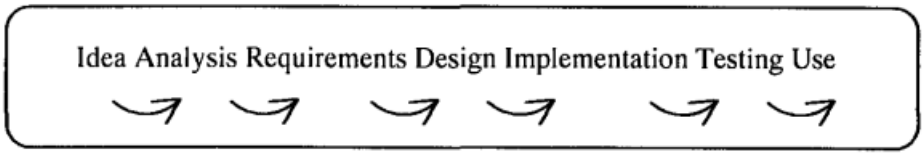
Attracting and Retaining Skilled Staff: The high costs of keeping systems operational and ensuring that user needs are satisfied - as implied by the quoted case surveys - are partly due to the problem of attracting and retaining highly skilled and talented software maintenance personnel. The historical reason for the lack of skilled staff and high personnel turnover was largely an image problem. Problems persist today for a variety of reasons.



Chapter 5:
The Maintenance Process

Software Production Process:

The software production process encompasses the whole activity from the initial idea to the final withdrawal of the system.



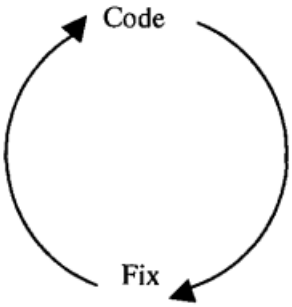
A process, being the series of discrete actions taken to effect a change, is distinct from the life-cycle which defines the order in which these actions are carried out. The software life-cycle starts with an idea, then goes through the stages of feasibility study, analysis, design, implementation, testing, release, operation, and use.

In software terms the model is the abstract representation of the software production process, the series of changes through which a software product evolves from initial idea to the system in use.

The term process implies a single series of phases. Life-cycle implies cycling through this series of phases repeatedly.

Critical Appraisal of Traditional Process Models:

1. Code and Fix Model:



It is a simple two-phase model. The first phase is to write code. The next phase is to 'fix' it. Fixing in this context may be error correction or addition of further functionality. Using this model, code soon becomes unfixable and enhanceable. There is no room in this model for analysis, design, or any aspect of the development process to be carried out in a structured or detailed way. There is no room to think through a fix or to think of the future ramifications, and thus errors increase and the code becomes less maintainable and harder to enhance. On the face of it, this model has nothing to recommend it.

Why then consider it at all? The reason is that despite the problems, the model is still used, the reason being that the world of software development often dictates the use of the code-and-fix model. If a correction or an enhancement must be done very quickly, in a couple of hours say, there is no time for detailed analysis, feasibility studies or redesign. The code must be fixed. The major problems of this scenario are usually overcome by subsuming code-and-fix within a larger, more detailed model. This idea is explored further in the discussion of other models. The major problem with the code-and-fix model is its rigidity. In a sense, it makes no allowance for change. Although it perhaps does not assume the software to be correct from the off - it does have a fix stage as well as a code stage - it makes no provision for alteration and repair. The first stage is to code. All the other stages through which a software system must

go (analysis, specification, design, testing) are all bundled together either into the fix stage or mixed up with the coding. This lack of properly defined stages leads to a lack of anticipation of problems.

2. Waterfall Model:

The traditional waterfall model gives a high-level view of the software life-cycle. At its most basic it is effectively the tried and tested problem-solving paradigm:

Decide what to do

Decide how to do it

Do it

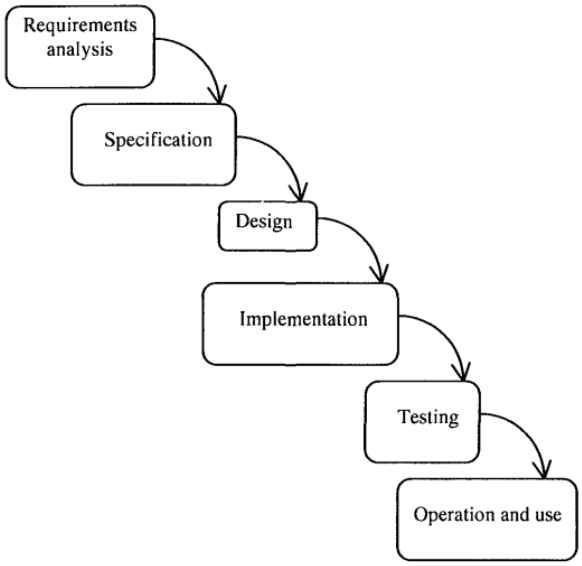
Test it

Use it.

The phases in the waterfall model are represented as a cascade. The outputs from one phase become the inputs to the next. The processes comprising each phase are also defined and may be carried out in parallel.

Many variations on this model are used in different situations but the underlying philosophy in each is the same. It is a series of stages where the work of each stage is 'signed off and development then proceeds to the following phase. The overall process is document driven. The outputs from each stage that are required to keep the process moving are largely in the form of documents. The main problem with the original waterfall model lay in its sequential nature, highlighted by later refinements which adapted it to contain feedback loops. There was recognition in this of the ever increasing cost of correcting errors. An error in the requirements stage for example, is far more costly to correct at a late stage in the cycle and more costly than a design error.

The model still fails to capture the evolutionary nature of the software. The model allows for errors in the specification stage, for example, to be corrected at later stages via feedback loops, the aim being to catch and correct errors at as early a stage as possible. However, this still assumes that at some point a stage can be considered complete and correct, which is unrealistic. Changes - in specification for example - will occur at later stages in the life-cycle, not through errors necessarily but because the software itself is evolutionary.



3. Spiral Model:

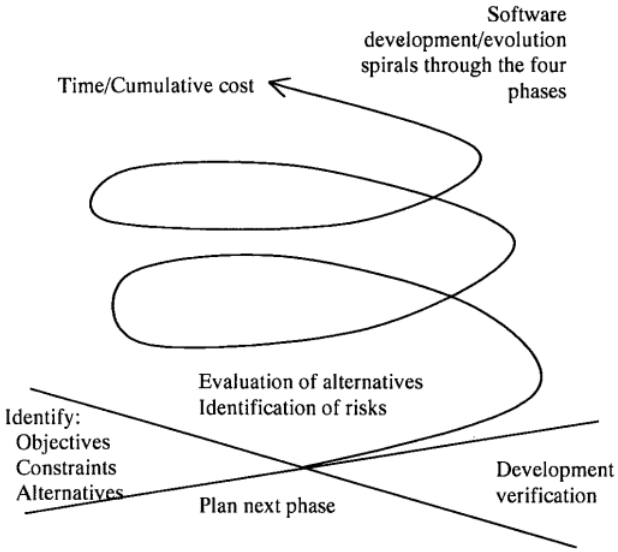
The phases in this model are defined cyclically. The basis of the spiral model is a four-stage representation through which the development process spirals. At each level

- objectives, constraints, and alternatives are identified,
- alternatives are evaluated, risks are identified and resolved,
- the next level of product is developed and verified,
- the next phases are planned.

The focus is the identification of problems and the classification of these into different levels of risk, the aim being to eliminate high-risk problems before they threaten the software operation or cost.
A basic difference between this and the waterfall model is that it is risk driven. It is the level of risk attached to a particular stage which drives the development process. The four stages are represented as quadrants on a Cartesian diagram with the spiral line indicating the production process.

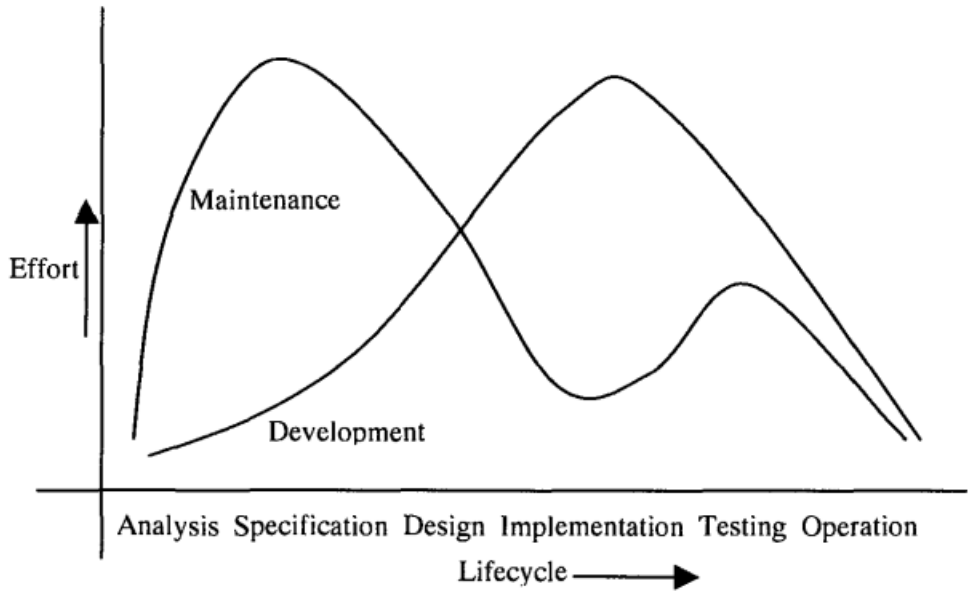
One of the advantages of this model is that it can be used as a framework to accommodate other models. The spiral model offers great advantage in its flexibility, particularly its ability to accommodate other life-cycle models in such a way as to maximise their good features and minimise their bad ones. It can accommodate, in a structured way, a mix of models where this is appropriate to a particular situation.

A problem with the spiral model is a difficulty in matching it to the requirements for audit and accountability which are sometimes imposed upon a maintenance or development team.
The spiral model requires that the high-risk areas are tackled first and in detail. Although 'difficult' does not always equate to 'high risk' it often does. A team inexperienced in risk assessment may run into problems.



Maintenance Process Models

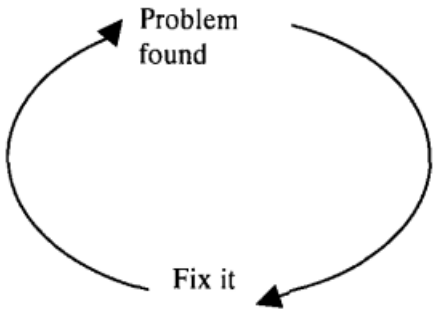
The generic stages in a maintenance-conscious model compared with the traditional development model appear similar on the surface but within the stages there are great differences in emphasis and procedure. There is more effort required and very different emphases on the early stages, and conversely less effort required in the later stages, of the maintenance model as opposed to the development model.



The essence of the problems at the heart of all the traditional models is in their failure to capture the evolutionary nature of software. A model is needed which recognises the requirement to build maintainability into the system. Once again, there are many different models and we will look only at a representative sample of four of them.

1. Quick-Fix Model

This is basically an ad hoc approach to maintaining software. It is a 'firefighting' approach, waiting for the problem to occur and then trying to fix it as quickly as possible, hence the name.



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 code structure. There would be little if any documentation. It is easy to see how the model emerged historically, but it cannot be dismissed as a purely historical curiosity because, like the code-and-fix model, it is still used.

In the 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.

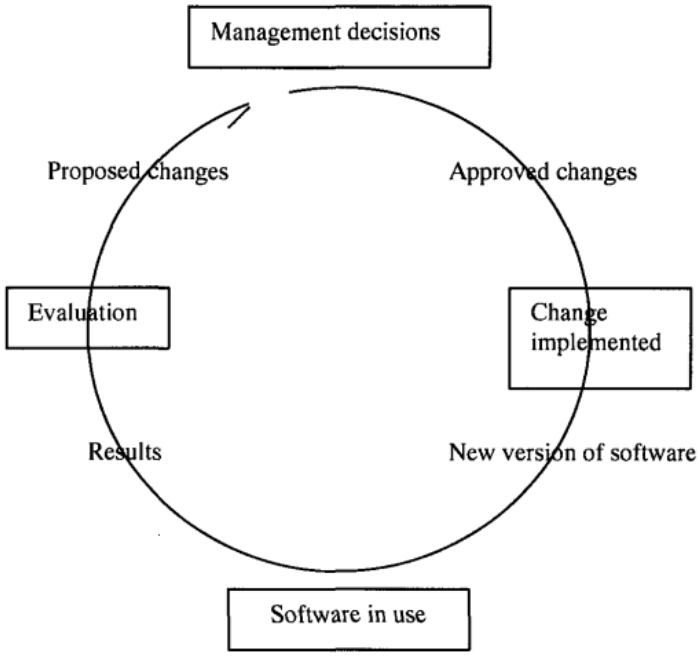
If an organisation relies on quick-fix alone, it will run into difficult and very expensive problems, thus losing any advantage it gained from using the quick-fix model in the first place.

The underlying basis of the problem is that the quick-fix model does not 'understand' the maintenance process. The problems experienced were not hard to predict and the 'advantage' gained by the original quick fix was soon lost.

2. Boehm's Model

In 1983 Boehm, proposed a model for the maintenance process based upon 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 maintenance but also help understanding of the process.

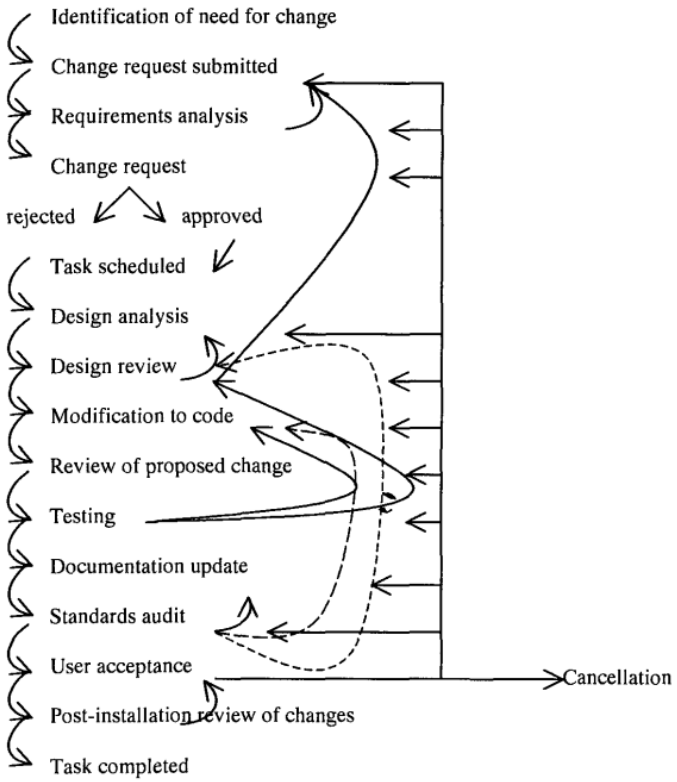
Boehm represents the maintenance process as a closed loop cycle. He theorises that it is the stage where management decisions are made that drives the process. In this stage, a set of approved changes is determined by applying strategies and cost-benefit evaluations to a set of proposed changes. The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



3. Osborne's Model

The difference between this model and the others described here is that it deals directly with the reality of the maintenance environment. Other models tend to assume some facet of an ideal situation - the existence of full documentation, for example. Osborne's model makes allowance for how things are rather than how we would like them to be.

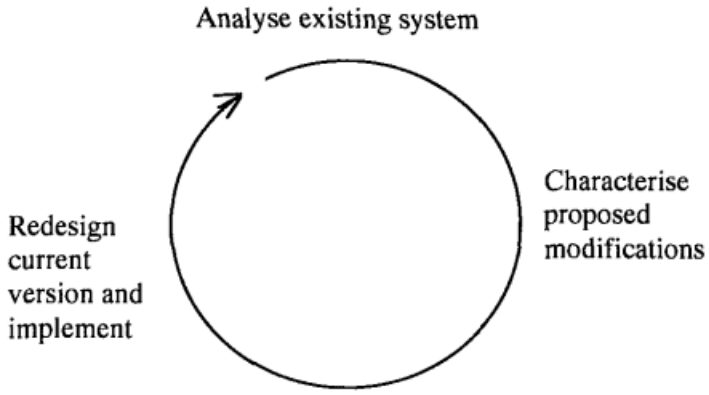
The maintenance model is treated as continuous iterations of the software life-cycle with, at each stage, provision made for maintainability to be built in. If good maintenance features already exist, for example full and formal specification or complete documentation, all well and good, but if not, allowance is made for them to be built in.



Osborne hypothesises that many technical problems which arise during maintenance are due to inadequate management communications and control, and recommends a strategy that includes:

- the inclusion of maintenance requirements in the change specification;
- a software quality assurance program which establishes quality assurance requirements;
- a means of verifying that maintenance goals have been met;
- performance review to provide feedback to managers.

4. Iterative Enhancement Model



This model has been proposed based on the tenet that the implementation of changes to a software system throughout its lifetime is an iterative process and involves enhancing such a system in an iterative way. It is like the evolutionary development paradigm during pre-installation.

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 for each iteration. The model is effectively a three-stage cycle:

- Analysis.
- Characterisation of proposed modifications.
- Redesign and implementation

The existing documentation for 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.

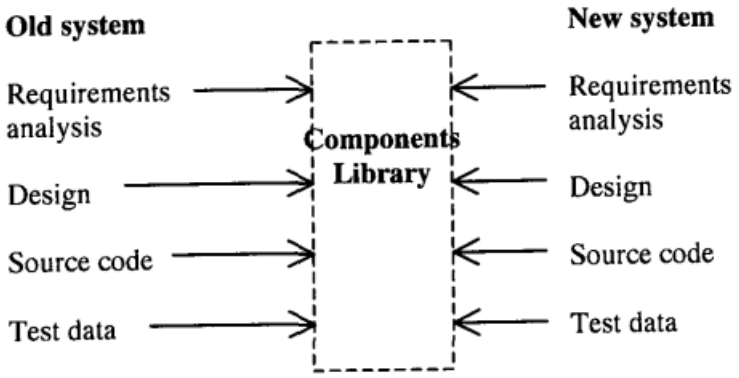
The problems with the iterative enhancement model stem from assumptions made about the existence of full documentation and the ability of the maintenance team to analyse the existing product in full. Whereas wider use of structured maintenance models will lead to a culture where documentation tends to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

5. 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 has four main steps:

- Identification of the parts of the old system that are candidates for reuse,
- Understanding these system parts,
- Modification of the old system parts appropriate to the new requirements,
- Integration of the modified parts into the new system.



When to make a change?

It cannot simply be assumed that everyone involved with a system, from the developers to the users, can throw their ideas into the arena and automatically have them implemented. That would lead to chaos.

Not all changes are feasible. A change may be desirable but too expensive. There must be a means of deciding when to implement a change. Ways of doing this, e.g., via a Change Control Board.

Process Maturity

Organisations need a means by which to assess the maturity and effectiveness of their processes.

Maturity Model:

The Software Engineering Institute (SEI) developed a capability maturity model for software. Using this, the maturity of processes can be assessed. Five levels are defined:

1. **Initial:** The software process is ad hoc. Few processes are defined. Success depends on individual flair of team members.
2. **Repeatable:** Basic processes are established, tracking cost, scheduling, and functionality. Successes can be repeated on projects with similar applications.
3. **Defined:** Processes are documented and standardised. There exists within the organisation standard processes for developing and maintaining software. All projects use a tailored and approved version of the standard process.
4. **Managed:** Detailed measures are collected, both process and the quality of the product. Quantitative understanding and control are achieved.
5. **Optimising**: Quantitative feedback is evaluated and used from the processes and from the piloting of innovative ideas and technologies. This enables continuous process improvement.

The SEI's model is not the only one in use, but is widely referenced and other models [63] tend to be closely cross-referenced with it. The benefits accruing from software process improvement based upon the SEI's model have been studied and documented.

Software Experience Base:

Four factors required for successful implementation of a software experience base:

1. **Cultural change -** people must become comfortable with sharing knowledge and using others' knowledge and experience, in order that the software experience base is active and used.
2. **Stability -** an unstable business environment will not be conducive to the development of a culture or a system for knowledge and experience sharing.
3. **Business value -** for any system to be used and useful in today's business world, it must provide a demonstrable payback.
4. **Incremental implementation -** implementing a software experience base in small increments is of use in keeping the process close to the users and, with effective feedback, prevents the software experience base becoming a remote and irrelevant entity.

All models have strengths and weaknesses. No one model is appropriate in all situations and often a combination of models is the best solution.

Chapter 6:
Program Understanding

Aims of Program Comprehension

The ultimate purpose of reading and comprehending programs is to be able successfully to implement requested changes.

Knowledge		Importance
1.	Problem domain	▪ To assist in the estimation of resources ▪ To guide the choice of suitable algorithms, methodologies, tools and personnel
2.	Execution effect	▪ To determine whether or not a change did achieve the desired effect
3.	Cause-effect relation	▪ To establish the scope of a change, to predict potential ripple effects and to trace data flow and control flow
4.	Product-environment relation	▪ To ascertain how changes in the product's environment affect the product and its underlying programs
5.	Decision-support features	▪ To support technical and management decision-making processes

Problem Domain

Being able to capture domain knowledge is now considered a far more important area than it used to be, this is partly because of the proliferation of computers in a wider spectrum of specialist problem areas.

In order to effect change or simply to estimate the resource required for a maintenance task, knowledge of the problem domain in general and the sub-problems is essential so as to direct maintenance personnel in the choice of suitable algorithms, methodologies and tools. The selection of personnel with the appropriate level of expertise and skills is another aspect. Information can be obtained from various sources - the system documentation, end users, or the program source code.

Execution Effect

At a high level of abstraction, the maintenance personnel need to know (or be able to predict) what results the program will produce for a given input without necessarily knowing which program units contributed to the overall result or how the result was accomplished. At a low level of abstraction, they need to know the results that individual program units will produce on execution. Knowledge of data flow, control flow and algorithmic patterns can facilitate the accomplishment of these goals.

For example, a specialist compiler programmer may want to know, at a higher level of abstraction, the output from a complete compilation process, and at a lower level, the output from the parser. During maintenance, this information can assist the maintenance personnel to determine whether an implemented change achieved the desired effect.

Cause-Effect Relation

In large and complex programs, knowledge of this relation is important in several ways.

- It allows the maintenance personnel to reason about how components of a software product interact during execution.
- It enables a programmer to predict the scope of a change and any knock-on effect that may arise from the change.
- The cause-effect relation can be used to trace the flow of information through the program. The point in the program where there is an unusual interruption of this flow may signal the source of a bug.

Product-Environment Relation

A product is a software system. An environment is the totality of all conditions and influences which act from outside upon the product, for example business rules, government regulations, work patterns, software, and hardware operating platforms. It is essential for the maintenance personnel to know not only the nature but the extent of the relation. This knowledge can be used to predict how changes in these elements will affect the product in general and the underlying programs.

Decision-Support Features

Software product attributes such as complexity and maintainability are examples that can guide maintenance personnel in technical and management decision-making processes like option analysis, decision-making, budgeting and resource allocation. Measures of the complexity of the system can be used to determine which components of the system require more resource for testing. The maintainability of the system may be used as an indicator of its quality.

Maintainers and their information needs

Members of the maintenance team - managers, analysts, designers and programmers - all have different comprehension or information needs depending on the level at which they function.

1. Managers:

Considering that one of the responsibilities of management is making decisions, managers need to have decision-support knowledge in order to make informed decisions. The level of understanding required will depend on the decision to be taken. For example, to be able to estimate the cost and duration of a major enhancement, knowledge of the size of the programs (in terms of lines of code or function points) is required. This estimate can then be used to determine whether it is more economical to replace the system with a vendor system. Managers do not necessarily have to know the architectural design of the system or the low-level program implementation details in order to carry out their duties.

2. Analysts

During maintenance, the analysts would be concerned with knowing how changes in this environment (for example, new government regulations or a new operating system) would affect the system. Thus, prior to implementing the change, the analyst needs to have a global view of the system, that is, a general picture of the interaction between the major functional units. The analyst is also required to determine the implications of change on the performance of a system.

During maintenance, the analysts would be concerned with knowing how changes in this environment (for example, new government regulations or a new operating system) would affect the system. Thus, prior to implementing the change, the analyst needs to have a global view of the system, that is, a general picture of the interaction between the major functional units. The analyst is also required to determine the implications of change on the performance of a system. environment, thereby assisting the analyst to gain a good understanding of the system without being distracted by low-level design or coding details.

3. Designers

The design process of a software system can take place at two levels: architectural and detailed design. Architectural design results in the production of functional components, conceptual data structures and the interconnection between various components. Detailed design results in the detailed algorithms, data representations, data structures and interfaces between procedures or routines. During maintenance, the designer's job is to:

- extract this information and determine how enhancements could be accommodated by the architecture, data structures, data flow and control flow of the existing system;
- go through the existing source code to get a rough idea of the size of the job, the areas of the system that will be affected, and the knowledge and skills that will be needed by the programming team that does the job.

The use of concepts such as information hiding, modular program decomposition, data abstraction, object orientation, and good design notations such as data flow diagrams, control flow diagrams, structure charts and hierarchy process input/output (HIPO) charts can help the designer obtain a good understanding of the system before designing changes.

4. Programmers

Maintenance programmers are required to know the execution effect of the system at different levels of abstraction, the causal knowledge and knowledge of the product-environment relation. At a higher level of abstraction (for instance, at the systems level), the programmer needs to know the function of individual components of the system and their causal relation. At a lower level of abstraction (for example, individual procedures or modules), the programmer needs to understand 'what each program statement does, the execution sequence (control flow), the transformational effects on the data objects (data flow), and the purpose of a set of program statements (functions)'

This information will assist the programmer in several ways:

1. To decide on whether to restructure or rewrite specific code segments;
2. To predict more easily any knock-on effect when making changes that are likely to affect other parts of the system;
3. To hypothesise the location and causes of error;
4. To determine the feasibility of proposed changes and notify management of any anticipated problems.

Comprehension Process Models

Programmers vary in their ways of thinking, solving problems and choosing techniques and tools. Generally, however, the three actions involved in the understanding of a program are: reading about the program, reading its source code, and running it.

1. Read About the Program

At this stage of the process, the 'understander' browses, peruses different sources of information such as the system documentation - specification and design documents - to develop an overview or overall understanding of the system. Documentation aids such as structure charts and data and control flow diagrams can be used. In many large, complex, and old systems (developed prior to the advent of good documentation tools, techniques, and practice), this phase may be omitted if the system documentation is inaccurate, out of date or non-existent.

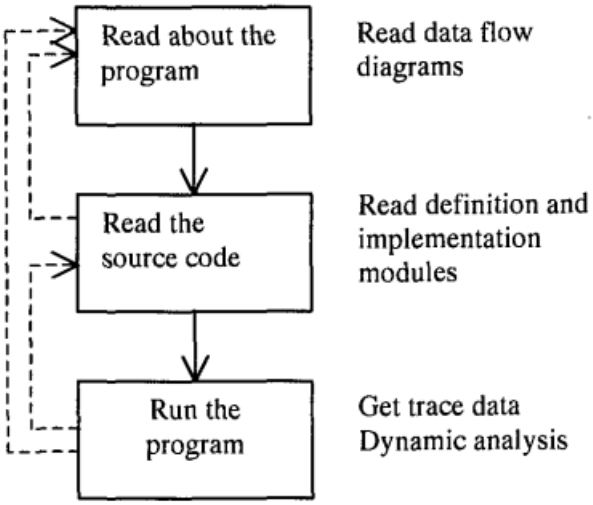
2. Read the source code

During this stage, the global and local views of the program can be obtained. The global view is used to gain a top-level understanding of the system and also to determine the scope of any knock-on effect a change might have on other parts of the system. The local view allows programmers to focus their attention on a specific part of the system. With this view, information about the system's structure, data types and algorithmic patterns is obtained. Tools such as static analysers - used to examine source code - are employed during this phase. They produce cross-reference lists, which indicate where different identifiers - functions, procedures, variables, and constants - have been used (or called) in the program. That way, they can highlight abnormalities in the program and hence enable the programmer to detect errors. Bearing in mind that the system documentation may not be reliable, reading program source code is usually the principal way of obtaining information about a software product.

3. Run the Program

The aim of this step is to study the dynamic behaviour of the program in action, including for example, executing the program, and obtaining trace data. The benefit of running the program is that it can reveal some characteristics of the system which are difficult to obtain by just reading the source code.

In practice, the process of understanding a program does not usually take place in such an organised manner. There tend to be iterations of the actions and backtracking (indicated by broken lines) to clarify doubts and to obtain more information.



It is generally assumed that as maintainers go through the steps outlined above (regardless of the order) to understand the program, they form a mental model - internal representation - of the program.

Mental Models

Our understanding of a phenomenon depends to some extent on our ability to form a mental representation, which serves as a working model of the phenomenon to be understood.

The phenomenon (how a television set works, the behaviour of liquids, an algorithm) is known as the target system, and its mental representation is called a mental model. For example, if you understand how a television works, then you have a mental model which represents this and, based on that model, you can predict behaviour such as what will happen when the television set is turned on or when a different channel is selected. Using the model, you can also explain certain observations such as the occurrence of a distorted image. The completeness and accuracy of the model depends to a large extent on its users' information needs. In the case of the television set, an ordinary user - who uses it solely for entertainment - does not have to understand the internal composition of the cathode ray tube and circuits and how they work, in order to be able to use it. A technician, however, who services the set in the event of breakdown

needs a deeper understanding of how the set works and thus requires a more elaborate and accurate mental model.

The content and formation of mental models hinges on cognitive structures and cognitive processes. The mental model is formed after observation, inference, or interaction with the target system. It changes continuously as more information about the target system is acquired. Its completeness and correctness can be influenced by factors such as the user's previous experience with similar systems and technical background. The mental model may contain insufficient, contradictory, or unnecessary information about the target system. Although it is not necessary for this model to be complete, it must convey key information about the target system.

Program Comprehension Strategies

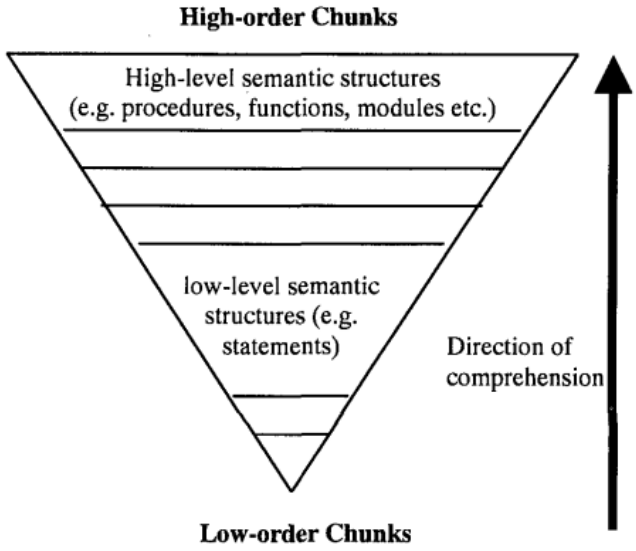
A program comprehension strategy is a technique used to form a mental model of the target program. The mental model is constructed by combining information contained in the source code and documentation with the assistance of the expertise and domain knowledge that the programmer brings to the task. Several descriptive models of how programmers go about understanding programs have been proposed based on the results of empirical studies of programmers. Examples of these models include top-down, bottom-up and opportunistic models.

1. Top-Down Model

The tenet of this model is that an understander starts by comprehending the top-level details of a program, such as what it does when it executes, and gradually works towards understanding the low-level details such as data types, control and data flows and algorithmic patterns in a top-down fashion.
Example: Brook's Model
The cognitive structure and cognitive process of a mental model resulting from a top-down strategy can be explained in terms of a design metaphor. Software development in its entirety can be a design task which consists of two fundamental processes - **composition** and **comprehension**. Composition represents production of a design and comprehension is understanding that design. Composition entails mapping what the program does in the problem domain, into a collection of computer instructions of how it works in the programming domain, using a programming language. Comprehension is the reverse of composition. It is a transformation from the programming domain to the problem domain involving the reconstruction of knowledge about these domains (including any intermediate domains) and the relationship between them. The reconstruction process is concerned with the creation, confirmation, and successive refinement of hypotheses. It commences with the inception of a vague and general hypothesis, known as the primary hypothesis. This is then confirmed and further refined on acquisition of more information about the system from the program text and other sources such as the system documentation.

2. Bottom-Up Model

Using this strategy, the programmer successively recognises patterns in the program. These are iteratively grouped into high-level, semantically more meaningful structures, The high-level structures are then chunked together into even bigger structures in a repetitive bottom-up fashion until the program is understood.



The chunking process tends to be faster for more experienced programmers than novices because they recognise patterns more quickly.

The main weaknesses of both the top-down and bottom-up comprehension strategies are:

- failure to take into consideration the contribution that other factors such as the available support tools make to understanding;
- and the fact that the process of understanding a program rarely takes place in such a well-defined fashion as these models portray. On the contrary, programmers tend to take advantage of any clues they come across in an opportunistic way.

3. Opportunistic Model

When using this model, the understander makes use of both bottom-up and top-down strategies, although not simultaneously.

According to this model, comprehension hinges on three key and complementary features - a knowledge base, a mental model, and an assimilation process:

- **A knowledge base:** This represents the expertise and background knowledge that the maintainer brings to the understanding task.
- **A mental model:** This expresses the programmer's current understanding of the target program.
- **An assimilation process:** This describes the procedure used to obtain information from various sources such as source code and system documentation.

When maintainers need to understand a piece of program, the assimilation process enables them to obtain information about the system. This information then triggers the invocation of appropriate plans from the knowledge base to enable them to form a mental model of the program to be understood. As discussed earlier, the mental model changes continuously as more information is obtained.

Factors that Affect Understanding

Several factors can affect not only the formation of mental models of a program, but also their accuracy, correctness, and completeness and hence the ease with which a program can be understood.

1. Expertise

Programmers become experts in a particular application domain or with a particular programming language by virtue of the repertoire of knowledge and skills they acquire from working in the domain or with the language. There is a psychological argument that this expertise has a significant impact on comprehension.

In effect, the more experienced a programmer is with an application domain or with a programming language, the easier and quicker it is to understand a program and indeed, the whole software system

2. Implementation Issues
There are several implementation issues that can affect the ease and extent to which a maintainer understands a program. The inherent complexity of the original problem being solved by the program is a factor. At the program level, the naming style, comments, level of nesting, clarity, readability, simplicity, decomposition mechanism, information hiding and coding standards can affect comprehension.

3. Documentation
The system documentation can be very useful in this respect, more importantly because it is not always possible to contact the original authors of the system for information about it. This is partly due to the high turnover of staff within the software industry: they may move to other projects or departments, or to a different company altogether. As such, maintainers need to have access to the system documentation to enable them to understand the functionality, design, implementation, and other issues that may be relevant for successful maintenance. Sometimes, however, the system documentation is inaccurate, out of date or non-existent. In such cases the maintainers must resort to documentation internal to the program source code itself - program comments.

4. Organisation and Presentation of Programs
The reading of program source code is increasingly being recognised as an important aspect of maintenance, even more so in situations where the program text is the only source of information about a software product. In the light of this, programs should be organised and presented in a manner that will facilitate perusal, browsing, visualisation and ultimately understanding.

Enhanced program presentation can improve understanding by:

- facilitating a clear and correct expression of the mental model of the program and the communication of this model to a reader of the program;
- emphasising the control flow the program's hierarchic structure and the programmer's - logical and syntactic - intent underlying the structure;
- and visually enhancing the source code using indentation, spacing, boxing and shading.

5. Comprehension Support tools
There are tools which can be used to organise and present source code in a way that makes it more legible, more readable, and hence more understandable.
Many comprehension tools are designed to serve as aids to enable the understander to speed up the understanding process. The output from these tools, however, does not provide explanation of the functionality of the subject system.

6. Evolving Requirements
In terms of requirements, we need to think in more flexible terms:

- Impact analysis is key, because changing requirements implies the addition, modification and deletion of requirements, which means that we have to introduce change as an integral part of building and maintaining systems.
- Written requirements cannot be treated as though written in-stone, just because they have been recorded. They will evolve.
- The evolution of requirements will lead to requirements that conflict. Resolution of conflict will become a part of requirements analysis.

- Requirements have always been an unofficial matter of negotiation. This will need to be recognised more formally.