University of Durham

School of Engineering and Computer Science

A Documentation Paradigm for an Integrated Software Maintenance Support Environment

David Bittlestone

Thesis submitted for the requirements of the degree

of

Doctor of Philosophy

1992

Abstract

Recent advances in computer hardware have not been matched by comparable advances in computer software, inhibiting the production of reliable software at greater levels of productivity.

Development of software is restricted by the so-called 'maintenance backlog'. Productivity in the maintenance sector has not kept pace with increasing annual labour costs, making the maintenance of software the major item in the budget of organisations responsible for the development and maintenance of software.

Gains in productivity can be anticipated by the exploitation of software-maintenance tools, within the framework of an Integrated Software Maintenance Support Environment (ISMSE), for which a high-level design has been proposed in this thesis, offering comprehensive support for *all* phases of the software life-cycle, particularly the maintenance phase.

A key factor in the reliable modification of software is the time taken to gain the prerequisite understanding, by a study of the system's documentation. This documentation degrades over a period of time, becoming unreliable, inhibiting maintenance of the software, which may be a large capital asset. Ultimately, the software may become impossible to maintain, requiring replacement.

Understanding gained during maintenance is wide-ranging and at various levels of abstraction, but is often NOT well-recorded, since no effective documentation system exists for recording the maintenance history of large software systems.

The documentation paradigm in this thesis, used within the framework of an ISMSE, aims to provide a means of recording the knowledge gained during maintenance, facilitating easier *future* maintenance, and preserving the reliability of the documentation, so reducing the time required to gain an understanding of the software being maintained. This provides a powerful means of increasing productivity, while simultaneously preserving a valuable capital asset.

Acknowledgements

I would like to thank my supervisors, David Robson and Malcolm Munro, for their help and encouragement, and for their comments upon numerous drafts of this thesis. I would also like to thank Colin Walter, Stephen Eldridge, and Greg O'Hare of UMIST for helpful discussions.

Financial support for this work was provided by a studentship from the Science and Engineering Research Council.

Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction to the Problem Area

1.1 The software maintenance problem

In recent years great advances have been made in computer hardware, but these have not been matched with advances in the productivity of reliable computer software; thus reducing the effectiveness of computer hardware. This state of affairs is due, in part, to the 'software maintenance backlog'; from this point onwards the term 'maintenance' is used in place of the term 'software maintenance'.

A *definition* of software maintenance is needed since it is the precursor to the establishment of a model of the maintenance process, the subject of the next chapter. Several definitions have appeared in the literature, each has its own merits: typical are those of IEEE [57] and Glass [46]. The IEEE [57] defines software maintenance as: 'Modification of a software product, after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.' Glass defines software maintenance as: 'The act of taking a software product that has already been delivered to a customer and is in use by him, and keeping it functioning in a satisfactory way. It is the process of being responsive to user needs, fixing errors, making user-specified modifications, honing the program to be more useful'.

The productivity of the maintenance programmer has not kept pace with increasing annual labour-cost, therefore the overall cost of maintaining software has grown to become, by far, the major item in the computing budget of most commercial organisations [70]. This means, in turn, that the development of new software has been restricted, exacerbating the growth in the gap between advances in computer hardware and the production of reliable software. This gap is referred to as the 'hardware-software gap' in the remainder of this thesis.

The traditional role of the maintenance programmer is concerned with manually scanning large amounts of printed material, seeking to assemble enough knowledge of a software system with the objective of making reliable modifications to it. This approach to software maintenance has proved inadequate in dealing with the maintenance problem and as software systems become larger and more complex with a greater anticipated lifespan (concomitant with the cost of development), automation of some aspects of the maintenance process offers

a means of decreasing the cost of maintenance through the increased productivity of the maintenance programmer.

Before any modification can be made to a software system a thorough understanding of the relevant part of the software is necessary because:

1. the place where the modification is to be made must be found
2. the change must be made without adversely affecting the software - i.e. the 'ripple effect' must be avoided.

Understanding is wide-ranging and at various levels of abstraction - a general understanding of the high level design may be needed to identify the modules which need to be modified. Partial knowledge of the control and data flows may be required to identify potential side-effects elsewhere in the application. A detailed knowledge of the implementation details in the vicinity of the change(s) is essential.

Program-understanding is the keystone of software maintenance, all other activities are subordinate, since without the necessary understanding of the software, the other maintenance phases cannot begin. The understanding of any system usually begins with a survey of the available supporting documentation, whose reliability for large complex software systems reduces with the increasing age of the software. For this reason, maintainers are often forced to rely on the source listing as the only reliable documentation of the software; however, the source listing only provides a low-level documentation aid, whereas the documentation which is most valuable to a maintainer is high-level documentation, such as design strategy

and functional specifications. The understanding that the maintainer seeks from the source listing has been referred to as the plans in the code by Letovsky et al. [67] and is initiated by the process of information-capture.

Traditionally, much of this information capture has been carried out by maintainers without the use of software tools: a labour-intensive task made even more difficult with a software system which has been in service for a considerable length of time, since the structure and readability of ageing software degrades with time. It has been shown [104] that a maintenance programmer spends most of his time in the analysis of code (about 47% of his time in total) and this labour-intensive activity reduces the productivity and effectiveness of the maintenance programmer because:

1. such a manual process is very time-consuming and error-prone
2. the onset of fatigue brought about by the drudgery of the task of manually scanning code to establish data and control flow, hinders the maintenance programmer in achieving the prerequisite understanding of the software
3. the knowledge gained during this analysis phase is often NOT recorded for the benefit of future maintainers because this aspect of software maintenance is not well-supported. This means that much time will be wasted in repetition of this understanding activity.

As a first step in reducing the maintenance backlog and narrowing the 'hardware–software gap', productivity-gains in the maintenance sector are of paramount importance. These gains in productivity can only be achieved by the introduction of automation, i.e. software tools,

and the adoption of a strategy for making use of automation, within the confines of a software maintenance support environment, thus releasing resources for software development.

Boehm [17] showed that the use of a software engineering environment can reduce development effort by 28 to 41%; since software maintenance is in some ways a microcosm of software development, it is hoped that a similar advantage will accrue through using a software maintenance environment. The GSA's Programmers' Workbench (PWB) [49] was the first initiative which attempted to achieve this increase in productivity and provided the impetus for the development of the Integrated Software Maintenance Support Environment (ISMSE), the basis of this research topic.

Although the environment is primarily intended to offer *technical* support to the maintenance programmer, through the creation and management of a repository of information needed to maintain software effectively, spin-off benefits are that the quality of maintenance will be raised and the *management* of maintenance will be made easier and more effective. The long-term goal of the ISMSE is to provide a maintenance team with an interactive environment, which will consist of an integrated comprehensive set of software maintenance tools and will make use of the latest database technology.

Before a maintenance environment can be created, it is necessary to have a comprehensive description of what activities take place within the maintenance phase and so the maintenance process must be accurately modelled; a maintenance model supplies knowledge about the role of the maintainer and the actions of the maintainer when maintaining software.

1.2 Thesis objectives and outline

1. To examine the need for a maintenance support environment and the need for a strategy for software maintenance

2. To investigate currently-available support environments for their support for software maintenance

3. To develop a strategy for the maintenance process and a high level design for an integrated maintenance support environment

4. To define formally and implement a paradigm for the documentation of maintenance and demonstrate and analyse its use

This chapter introduces the reader to the problem area. Chapter two surveys the maintenance models in the literature to see if there are any gaps. The design of an environment is based on a model of the process it is to support, and so the description of the maintenance process must be a comprehensive one.

Chapter three considers integrated software engineering environments to see whether a disciplined application of software engineering techniques is likely to be of help in the maintenance of software, particularly with regard to improving the productivity of a maintenance organisation.

Chapter four surveys some integrated software engineering environments and evaluates their support for the maintenance of software.

Chapter five sets the scene for the remainder of the thesis, indicating specifically where the ISMSE can be of help to a hypothetical generalised maintenance organisation, using the maintenance model derived in an earlier chapter. Deciding the role of the ISMSE determines the direction of the research.

Chapter six is concerned with the high-level design of a software maintenance support environment, highlighting the essential differences between the proposed ISMSE and current Integrated Project Support Environments (IPSEs).

Chapter seven is concerned with the information structure which is designed to hold information concerning the maintenance of a software system. Attention is focussed on the toolset and the role that the toolset plays in the ISMSE, i.e. capture of information, and the processing of information into knowledge, to aid in the understanding of software, prior to its maintenance. To aid in the *future* maintenance of software, particularly as regards the understanding of the software, and the associated improvement in productivity, a documentation paradigm is proposed for a maintenance history, which gives a suggested direction for a partial implementation of the ISMSE.

Chapter eight is concerned with the formal specification of the Information Structure as an ADT, using algebraic axioms.

Chapter nine prototypes a subset of the ISMSE, describing a Prolog implementation of the documentation paradigm.

Chapter ten is concerned with the testing of the validity of these specifications and the sim-

ulation of the environment and its maintenance strategy, mapping the Maintenance History and its associated editing functions onto the Unix file structure and toolset.

Chapter eleven relates to the success of the work, discusses the results of the thesis, and makes suggestions for further work.

Summary

Because of the software maintenance backlog, increases in productivity are of urgent and immediate priority and may be facilitated by:

1. The partial automation of the maintenance process, in particular to make the task of program-understanding easier through the capture of information and its subsequent management.

2. The provision of an environment in which the *best* use is made of automation: although it may be thought that an increase in productivity is a natural consequence of automation, the output from some tools is copious. Without controls, the volume of information they produce may even have a negative effect on productivity, highlighting the need for abstraction. At present the task of interpreting information output from tools is largely a manual one, and scanning large amounts of printed matter is a time-consuming and potentially error-prone process.

3. Structuring the recording of the maintenance process brings the maintenance process under control and in addition makes it easier to keep track of the progress of a maintenance assignment, thus saving time and increasing productivity.

4. Studying current methods, leading to suggestions for improvements to these methods.

It is to be hoped that, in addition to providing gains in productivity, an ISMSE can point the way to the establishment of a strategy for the maintenance of software and act as a test-bed for future initiatives in software maintenance research.

Chapter 2

Maintenance models

2.1 The role of the maintenance model

The maintenance process model is a description of the activities of a maintainer or maintenance organisation, from the receipt of the change request until the release of the new version of the software. The process model is a prerequisite for obtaining a high-level view of the overall requirements for the ISMSE, since, as pointed out by Stenning [120], the role of an environment is to support the effective use of an effective process. The high-level requirements for the ISMSE are the subject of chapter six.

2.2 Literature survey of maintenance models

2.2.1 Introduction

A number of maintenance process models have appeared in the literature; these are now surveyed and the chapter is concluded by a discussion of the models. An assessment of the strengths and weaknesses of the models is used to suggest a generalised model of the maintenance process which can be used to derive the high level requirements for the ISMSE.

The development of models of the software maintenance process is examined from a historical perspective; the models discussed are subdivided into two classes, those which provide a high-level view of the software maintenance process, and those which provide a more detailed, lower-level view; within each class the models of the maintenance process are examined in order of their chronology.

Models due to Boehm (1976) [15], Liu (1976) [76], Sharpley (1977) [116], Mellor (1986) [83] offer a high level view of maintenance; others due to Belady (1976) [8], Parikh (1982) [100], Yau (1982) [130], Martin (1983) [79], Patkau (1983) [102], Osborne (1983) [96], Carter (1986) [24], An [5] and Chapin (1988) [25] offer a more detailed lower-level view.

The models due to Belady [8], and An et al [5] are derived from code-level views of software systems. The model due to Belady and Lehman (1976) [8], is based on empirical observations of several large software systems and is concerned with the evolution of software. An et al [5]

propose a model of the maintenance process based on the changes that occur in software as it is maintained, a pattern of changes being used to distinguish between types of maintenance. These models lie outside the scope of this survey.

2.2.2 The models

2.2.2.1 Boehm model

According to Boehm [16] maintenance can be decomposed into three phases, and this decomposition is now generally accepted. These phases are:

1. Understanding

   Good documentation and traceability between requirements and code are needed, with well-structured and well-formatted code.

2. Modification

   Software and hardware and data structures should be easy to expand and should minimise the side-effects of changes; easy-to-update documentation is needed.

3. Validation

   Software structures should facilitate selective retesting, and aids for making retesting more thorough and efficient are needed.

Boehm offers no further refinement of the model. In particular, no further guidance is offered as to how to proceed should these desirable characteristics of software be absent.

2.2.2.2 Liu model

Liu does not refer to a model as such but describes the 'maintenance function', as follows.

1. The capacity, function and logic of the existing program or system must be thoroughly understood.
2. New logic is developed to reflect the new request or additional feature.
3. The new logic must be incorporated into the existing one.
4. Ensure that the new logic is functionally correct, and that the unmodified portions of the system are not inadvertently affected or disturbed. This last point is concerned with the ripple effect and Liu emphasises the need for testing.
   (a) Test for system failure first
   (b) Test the unmodified portion of the system
   (c) Test the modified portion with all imaginable conditions
   (d) Aim at the few most representative situations which constitute a major portion of the system
   (e) Test the *documentation* of the changes made to the program, as well as the program itself.

Liu offers no help in deciding how the understanding and modification phases should be carried out, but his last point, (e), has important consequences for the increasing complexity of ageing software, and its subsequent degradation. This is referred to again, in the discussion at the end of this chapter.

2.2.2.3 Sharpley model

Sharpley [116] restricts his model to the area of corrective maintenance which he decomposes into four discrete phases:

1. Verification of the problem - reproduction of error symptoms and attribution of the error to software, hardware, or the interface between the two.
2. Diagnosis of problem and isolation of the part of the system responsible for the error.
3. Re-programming and regeneration of the system.
4. Baseline validation - establishment of correct operation.

The scenario here is that of a team of highly-skilled people, on standby, completely familiar with the embedded software-system, in a 'high-technology', 'high-risk' industry, cast in the role of 'firefighters', in case of abnormal behaviour of the program. There is no 'learning curve' here, and so there is not the same emphasis on the understanding phase as in a typical maintenance assignment in the commercial sector.

2.2.2.4 Mellor model

Mellor [83] defines a failure as 'Non-conformance between actual product behaviour and the specified behaviour', and notes that ...'failures in a product are often user-specific, since they are specific to a customer's usage and different customers use different parts of the same product, and to different extents. A priority is assigned to fixes, a fault which has trivial effects and is difficult to fix, is assigned a low priority, and vice-versa.' Mellor concentrates on the problem-verification phase of corrective maintenance and how a maintenance department can best be structured to cope with maintenance requests from users. No other model looks at this aspect of maintenance at the same level of detail.

Mellor classifies two types of problems with software:

1. Usability problems: These are due to a error in the original requirements definition, the product conforms to specification, but there is a feature which causes problems in use. A usability problem may be irreparable, and the only remedy is an 'avoidance action'.

2. Problems due to errors in the code which mean that the specifications have not been met. He also makes the point that a fault in the user manual is also a fault in the product.

2.2.2.5 Parikh model

The maintenance task is decomposed into four phases:

1. Identification of objects

   The specifications or enhancements of the maintenance request are reviewed. All personnel concerned with the request are consulted.

2. Understanding the software

   An inventory of the affected program is taken with the associated documentation. The affected program is investigated.

3. Modification of code

   The areas in the code where the modification is to be made are located. Possible ripple effects are checked for as the result of the new design. The new changes are coded and implemented.

4. Validation of the modified program

   A walkthrough of the changes in the modified program is made and then the modified program is tested. Review the test results and put the program into production. Update the relevant documentation and conduct post-test reviews.

Parikh's model is offered in the context of time estimates and the model seems to be restricted to enhancements, since the modification of the code is to cope with a new design.

2.2.2.6 Carter model

Carter's model of corrective maintenance contains seven phases.

1. Problem detection

   Detection of significant difference between expected output and actual output, usually by the user.

2. Problem determination

   Recognition of symptoms which constitute abnormal behaviour on the part of the system.

3. Diagnosis (Understanding)

   The maintainer ingests system data and produces more data of his own, assimilates documentation and determines which part of the system is causing the problem.

   This stage also produces information about the scope of the effects of the problem.

4. Correction and testing

   Code is added or changed (rarely deleted), or data is patched or deleted. Testing of the correction occurs here.

5. Recovery

   Corrected code or data is installed, files are rolled back or updated to cover the effects of the problem, and the production stream is restarted at whatever point is necessary.

6. Reporting

   Maintenance managers, users and their management need to be informed as to what the problem was, and why it occurred - its symptoms, characteristics, its resolution, and any analysis the maintainer can produce must be archived for future reference.

7. Review

   The diagnosis of the problem and the solution is reviewed critically in an attempt to

validate, by expertise, individual and collective standards and techniques. Experience can be shared between more and less experienced personnel, and harmful or ineffective techniques can be corrected in a technically valid setting.

Carter restricts his view of software maintenance to that of 'response maintenance', i.e. corrective maintenance. This is the only aspect of maintenance that he considers, perfective and adaptive maintenance are not addressed. In common with Parikh, Carter emphasises the need for a post-test review.

2.2.2.7 Chapin model

He subdivides the phases of maintenance into a series of steps as shown below.

1. Understand existing system

   Personnel review any existing documentation and access relevant materials and personnel who may possess relevant knowledge.

2. Define the objectives for the modifications

   The maintainer seeks to clarify the aspirations of the user in requesting the change to the program.

3. Analyze the requirements

   The consequences of exploring alternate paths in satisfying the maintenance request are considered and evaluated with an accompanying cost-benefit analysis.

4. Specify modification(s) to be made

   A summary of the analysis results from the previous step produces a specification for the proposed modification.

5. Design modification(s)

6. Program modification(s)

7. Code and compile

8. Debug and test

   The testing aims to prove that the appropriate change has been correctly implemented.

9. Revalidate

   This attempts to confirm the stability of the system. i.e. those parts of the system which were *not* intended to change have not done so.

10. Train users prior to release of new software

    As soon as the specification step is completed the users are trained to use the modified system to gain familiarity prior to its release

11. Convert from previous version of software and release

    The author does not specify this any further.

12. Document and perform Quality Assurance review

    This process is performed concurrently with the above steps and provides the basis for inspections, walkthroughs, technical and management reviews.

Presumably steps 5-7 follow an iterative process, although the author does not specify further. Only Chapin explicitly refers to retraining users in the use of the updated system. Chapin's model [25] lacks a problem-verification phase and it is not clear whether his model applies to all types of maintenance.

2.2.2.8 Martin-McClure model

In common with other models the high-level tasks are:

1. Understanding
2. Modification
3. Revalidation

Each of these three phases is further decomposed [79] as described below.

1. **Understanding**

   This is broken down into:

   (a) Top-down comprehension

      i. The need to become familiar with the overall program purpose and the overall flow of control.

      ii. Identify the basic program structures as well as the processing components.

      iii. If the program is part of a larger system then delineate its role.

iv. Identify what each component does and how this is implemented in the code.

(b) Improvement of documentation.

i. As understanding of the program is gained, document it in a high level fashion.

ii. Participation in program development

The maintainer-to-be should take part in the development of the program.

2. **Modification**

(a) Design the change and debug

If the change is an error then this is rectified by changing the program logic. If the change is an enhancement then new logic is developed and incorporated into the program

The design of the new logic is top-down:

i. Review entire program at a general level by studying modules, their interfaces and the database.

ii. Then isolate the modules and the data structures which are to be changed and those modules and data structures which are affected by the change.

iii. Detailed study of module and data structures, design change, specifying new logic and changes (if any) to existing logic.

(b) Alter code

Changes should be implemented as simply as possible, exercising caution and preserving existing coding style.

(c) Minimise side-effects

i. Search all modules which share global variables or routines with the changed module.

ii. When multiple changes are envisaged the changes should be grouped by module. The sequence of changes should follow a top-down approach, changing the main driver first, then its direct descendants and so on.

iii. Change one module at a time, determining potential ripple effects, before changing the next module in the sequence.

3. **Revalidation**

Revalidation is necessary to ensure that the modifications carried out to the program have not adversely affected the program. Revalidation is achieved by carrying out testing, each type of testing having its own particular goal.

(a) System testing

Does the program work as before ?

(b) Regression testing

Have the changes affected how the rest of the program works ?

(c) Change testing

Have the changes been designed and implemented correctly ?

Martin's model is a comprehensive one and offers detailed guidance for all phases of the maintenance process.

2.2.2.9 Patkau model

The five basic maintenance tasks are identified in a high-level manner.

1. Identification and specification of the maintenance requirements.
2. Diagnose and change location
3. Design of the modification
4. Implementation of the modification
5. Validation of the new system

There are four possible types of maintenance modification:

1. Corrective
2. Enhancement
3. Adaptive
4. Perfective

Steps 1-3 differ according to the type of modification, steps 4-5 are as for Parikh and Martin-McClure models. Patkau identifies four possible kinds of maintenance modification.

1. **Corrective**

(a) Identify repeatable error symptoms and specify the correct operation of the system - for this a test system and test data are needed.

(b) Locate the part of the system responsible for the error.

(c) Design the desired properties of the system, after deciding what they should be. Determine the side-effects of the changes in these properties.

2. **Enhancement**

(a) Identify new or altered requirements and specification of the operation of the enhanced system.

(b) Locate the existing elements affected by the enhancements.

(c) Design is split into the following sub-tasks.

i. Assess how new requirements could be met by modifying existing components.

ii. Decide what new components are required

iii. Develop the specifications of the new components and/or revise the specifications of existing components.

iv. Examine the side-effects of the revised specifications and/or the addition of new components

3. **Adaptive**

(a) Identify the type of change in the processing or data environment, describe the change and revise all specifications to reflect the change.

(b) Locate all software elements affected by the change. When there is a change in the data environment locate the parts of the system which use or set the data

being changed. Use a data dictionary to store the system inputs and outputs, where they are used and their properties.

(c) Design can be accomplished by employing techniques used for corrections or enhancements, for changes in the data environment and minor changes in the processing environment.

4. **Pefective**

(a) Identify a deficiency in the performance, quality, standards, maintainability, specify the change in performance or quality standards.

(b) Locate the sources of the deficiencies.

(c) Design entails some re-design of a portion of the software such that it still satisfies the original requirements, but the new software either:

i. Uses less resources

ii. Is coded or structured better

iii. Is more maintainable

iv. Is a combination of all three.

Patkau's model is a comprehensive one and offers detailed guidance for all phases of the maintenance process.

2.2.2.10 Osborne model

Osborne [96] models the maintenance process as:

1. Determination of need for change
2. Submission of change request
3. Requirements analysis
4. Approval/rejection of change request
5. Scheduling of task
6. Design analysis
7. Design review
8. Code changes and debugging
9. Review of code changes
10. Testing
11. Update documentation
12. Standards audit
13. User acceptance
14. Post installation review of changes and their impact on the system
15. Completion of task

There are a number of iterative steps within the model, for example the change request may be referred back to the user for clarification.

2.2.2.11 Yau model

Yau models the maintenance process as distinct phases:

1. Determining the maintenance objectives

   (a) Correct program errors

   (b) Add new capabilities

   (c) Delete obsolete features

   (d) Optimisation

2. Understanding the program

   The ease of understanding is affected by:

   (a) Complexity

   (b) Documentation

   (c) Self-descriptiveness

3. Generating maintenance proposals

   The proposed alterations to the system are affected by the extensibility of the program.

4. Accounting for ripple effect

   This is affected by the stability of the program which Yau defines as 'The resistance to the amplification of changes in the program.'

5. Testing

   This is affected by the testability of the program - testability is not defined. If the

testing of the program is not successful then the maintenance process is performed iteratively.

The model represents information about the development and maintenance of software systems, emphasising relationships between different phases of the software life cycle, and provides the basis for automated tools to assist maintenance personnel in making changes to existing software systems.

2.2.3 Other contributions to modelling the maintenance process

2.2.3.1 Introduction

The description of maintenance process models would not be complete without a reference to the underlying psychology, particularly with respect to program understanding.

2.2.3.2 Fjeldstad and Hamlen's contribution

Fjeldstad and Hamlen [39] found that in a study concerning program enhancement, expert maintenance programmers spent as much time understanding the program as they did constructing the enhancement. In addition, the same programmers studied the original program about three and a half times as long as they studied the associated documentation. These facts demonstrate the importance of program understanding and suggest that maintenance

programmers regard the source code itself as the most reliable documentation concerning the program.

2.2.3.3 Littman et al's contribution

Littman et al [75] showed that, commonly, there are two strategies adopted by maintenance programmers for program understanding; the 'as-needed' strategy, where the maintainer studies and understands only as much of the program as is necessary to carry out the maintenance task, and the 'systematic-strategy', where the maintainer seeks to develop a global understanding of the program. For small programs, the 'systematic-strategy' was found to be superior, but for large programs maintainers were forced to adopt the 'as-needed' strategy, since the complexity of large programs exceeds the mind's capacity to understand them, in a reasonable time-scale.

2.2.3.4 Brooks' contribution

Brooks [20] asserts that understanding a program concerns:

1. What each statement means
2. How flow of control passes from one statement to another
3. What algorithms have been employed
4. How information is represented and transformed in data structures

5. Which programs invoke other sub-programs

6. How the program interacts with its environment

The above has been described by Brooks as a succession of knowledge domains that bridge between the problem domain and the executing program [20]. A knowledge domain is a collection of information about objects of some sort and relationships between those objects. The process of understanding a program is one of constructing or reconstructing the knowledge domains and relations among them from the code, comments and whatever other documentation is available. Brooks' model makes inferences about documentation - e.g. languages like FORTRAN require more explanation of their code than languages like Pascal, which allow direct manipulation of higher-level abstractions. The layered structures of knowledge of a program is provided by abstraction.

2.2.3.5 Schneiderman and Meyer's contribution

Scheiderman and Meyer [114] propose a syntactic/semantic model of program behaviour; the model assumes that semantic and syntactic knowledge is stored in long-term memory and manipulated in short-term and working memory. They suggest that program comprehension is mainly building up a hierarchy of semantic knowledge about what the program does at the top of the hierarchy, and lower-level information about statements and algorithms below. The representation is in terms of abstractions, e.g. representing the function groups of statements, derived from the text.

2.2.3.6 Letovsky's contribution

Letovsky [66] suggests that when a reader has a complete understanding of a program he possesses a description of the goals of the program, the actions and data structures of the implementation, and an explanation of how goals or sub-goals are accomplished by the components of the implementation. The reader's hierarchy is one of goals and sub-goals.

2.2.3.7 Letovsky and Soloway's contribution

The work of Letovsky and Soloway [67] on delocalised plans and program comprehension, defined the following terms:

1. Algorithm - conceptually different from a plan
2. Goal - denotes intentions.
3. Plan - denotes techniques for realising intentions.

The conclusions of Letovsky and co-workers were:

1. Program understanding is recognising plans in the code. If the lines which implement a plan are delocalised then the plan is difficult to follow.
2. Programmers often guess the intention of a plan from the first few remote lines of code without bothering to read ahead in the program - false assumptions are often made,

which usually means that a program becomes incorrectly modified (this is most likely to happen when verification is difficult, the plausibility of the assumption is high and the perceived importance of the assumption is small); this can be reduced by the use of program-analysis tools.

3. The goal of a variable is different from its role.

4. Programs become more complex with each modification because maintainers are reluctant to delete any existing code on the basis that their understanding of the program may be incomplete.

5. Each plan in the program requires a documentation entry which indicates the purpose of the plan and the proposed implementation. The entry should contain pointers to the relevant lines of code.

All of the preceding models involve layers of knowledge that becomes progressively more abstract and that are ultimately tied to larger and larger fragments of the program.

2.2.3.8 Linger, Mills and Witt's contribution

Two characteristics of a program have a dominant influence on reading approaches that may be available - the degree of documentation and whether the program is structured. Linger, Mills and Witt [71] point out that poorly-documented code generally must be read bottom-up, as the lack of documentation can make it nearly impossible to devised hypotheses about what various sections of code accomplish, without examining those sections of code in detail

- well-documented code can usually be read top-down

Models of program understanding may offer guidance in the choice of software tools for an ISMSE, or may be instrumental in producing ideas for new tools, or in suggesting new uses for *existing* tools, e.g. understanding plans in the code may be facilitated by tools such as Weiser's [127] program slicer.

2.3 Discussion

2.3.1 Characterisation of software maintenance

Swanson's [121] and Boehm's [15] characterisations (1976) of software maintenance were partly responsible for refocusing the spotlight on this subject area, which hitherto had been somewhat neglected. This followed Boehm's report (1973) [14] that almost 40% of the software effort went into maintenance.

Boehm [16] characterised software maintenance into two main types.

1. Software update - functional specification is updated (Enhancement).

2. Software repair - functional specification is unchanged (Corrective).

Swanson (1976) characterised maintenance as being of three main types.

1. Corrective

   In this category he includes 'bugs' in the software, failure to meet performance criteria as regards functional specifications, failure to meet programming standards set by the organisation, or inconsistencies or incompleteness in the detailed design, derived from the functional specifications.

2. Adaptive

   This is in response to a change in the data environment or the change in the processing environment.

3. Perfective

   This is 'Maintenance performed to eliminate processing inefficiencies, enhance performance, or improve maintainability. Its aim is to make the program a more perfect design implementation.'

Swanson's characterisation of maintenance has received wide acceptance, but there is some disharmony as regards major enhancements to software, which accounts for 60% [70] of software maintenance. A major enhancement to software is generally accepted as belonging to the category of perfective maintenance.

Some authors describe perfective maintenance as 'fine-tuning' which is in broad agreement with Swanson's definition above, but other authors e.g. Patkau [102], Belady [8] regard major enhancements to software as being distinct from perfective maintenance, as defined

by Swanson. This divergence appears to be in response to scale. A major enhancement is not concerned with 'a more perfect design implementation', or 'fine tuning'; rather it is concerned with substantial alterations to the original design to take into consideration the changing needs of an organisation (and therefore changing requirements of the software): major enhancements to software take software maintenance back into the realm of software development. It seems, however, to be accepted by many software practitioners that since enhancement is a *post* development activity, i.e. it comes *after* the release of the software, then it is maintenance. Furthermore since enhancement clearly does not belong to *either* category of corrective or adaptive maintenance as defined by Swanson, then it must belong to the third category, i.e. perfective maintenance.

This is inherently unsatisfactory, since as Swanson [121] points out ... 'Without making some important distinctions between types of maintenance activity undertaken, it will be impossible to discuss the effective allocation of these activities toward organisational ends.' No *distinction* has consciously been made as to where to place major enhancements to software, it seems to have found its niche by default.

The IEEE (1987) [57] glossary of software engineering terminology defines perfective maintenance as 'Maintenance performed to improve performance, maintainability, or other software attributes' (attributes are not specified). Performance is defined as 'The ability of a computer system to perform its functions, e.g. response time, throughput, number of transactions.' Again, this would seem to exclude major enhancements of software from Swanson's category of perfective maintenance, the IEEE definition is more redolent of 'fine tuning.' This suggests that another category of software maintenance is required, perhaps *re*development ?

Another type of software maintenance which has recently come to the fore is 'preventive maintenance' [9, 105], which may be defined as 'maintenance performed on a scheduled basis before the manifestation of any deficiency in the working of the software'. The aim of preventive maintenance is to reduce the future maintenance effort, by the introduction of forward planning with regard to the work of the maintenance organisation, instead of relying on 'crisis management' as a substitute for strategy. However, preventive maintenance has not yet gained universal acceptance, neither has its efficacy been established.

2.3.2 Assessment of the the maintenance models surveyed

Those models which attempt to provide a full description of the technical aspect of the maintenance process, due to Boehm (1976) [15], Liu (1976) [76], Sharpley (1977) [116], Parikh (1982) [100] Yau (1982) [130], Martin (1983) [79], Patkau (1983) [102], Osborne (1983) [96], Carter (1986) [24], and Chapin (1988) [25] take the same high-level view, i.e. they all identify three phases:

1. Understanding
2. Modification
3. Revalidation

Only Mellor [83] *explicitly* focuses on the area of problem-verification, which may preclude the need for maintenance, since solutions may already have been provided for 'new' problems.

	Category of Maintenance															
	Corrective				Adaptive				Perfective				Enhancement			
	V	U	M	T	V	U	M	T	V	U	M	T	V	U	M	T
Boehm		/	/	/		/	/	/		/	/	/		/	/	/
Liu		/	/	/												
Sharpley	/	/	/	/												
Parikh													/	/	/	/
Yau	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
Martin		/	/	/		/	/	/		/	/	/		/	/	/
Patkau	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
Osborne	/	/	/	/	/	/	/	/	/	/	/	/				
Carter	/	/	/	/												
Chapin		/	/	/		/	/	/		/	/	/		/	/	/
Mellor	/				/				/				/			

Key	
V	Verifying need for maintenance
U	Understanding
M	Modifying
T	Testing

Table 2.1: Summary of Maintenance models.

These models are summarised in table 2.1 above. The particular phase of the technical aspect of maintenance dealt with by each author is denoted by a diagonal line in the appropriate box.

In general, the models surveyed fall naturally into two categories.

1. From the viewpoint of *people*, i.e. those things which are done *by* people *to* software.

2. From the viewpoint of the *software*, i.e. those things which are done *to* the software *by* people, and how this affects the software.

From the point of view of designing a maintenance environment, the models from the first category are most useful. The reason for this is that when designing a maintenance environment the designer must imagine himself in the position of a maintainer. A maintainer asks

himself questions such as 'What must I know before I can alter this code ?', and 'What else must I change if I make this change to the program ?' The maintainer instigates change, and feels *actively* involved. The second category of model places the maintainer in a *passive* role.

Modelling relies heavily on the process of abstraction, removing unwanted detail that obscures the underlying fundamental principles involved, and postponing those judgements concerning the nature of a process that cannot be verified until the basic principles concerning that process are better understood. Examples of such fundamental principles include those relating to the technical aspects of maintenance which involve creativity; these are the most difficult to model, which may account for the fact that most of the models surveyed do not describe in detail the understanding and modification phases of software maintenance. These are the phases of maintenance where refinement of the model leads to divergence of approach, since here no two maintainers will use exactly the same model, because of the human factor involved. By contrast, the verification and validation phases are most independent of the human factor, since well-established procedures exist for establishing the veracity of, for example, functional specifications, or the response of the program to test cases. Perhaps with this is mind, some of the earlier attempts at deriving a maintenance model limited themselves to a high-level view, and this view has received general acceptance in the academic and DP communities.

This approach of using abstraction to establish a generalised maintenance model will enable maintainers to assemble and combine tools in ways which are not as yet predictable. On the other hand, more refined views of the maintenance model are useful because they indicate

the type of software tools that will be needed to automate partially the maintenance process and the type of information concerning software that needs to be stored. There exists an 'abstraction-threshold' beyond which important model-infrastructure is lost, so a balance must be struck which makes it possible to reconcile the conflicting demands of abstraction and refinement.

2.3.3 Establishing a generalised maintenance process-model

The exact description of the maintenance process is a function of the age of the software, since software (especially documentation) degrades with increasing age. For example, Belady's work [8] shows that a program's structure tends to degrade because of 'patches' inserted to fix bugs. In an extreme case, the understanding phase, which takes up most of the time in a maintenance assignment and relies heavily on documentation, will assume an even greater importance. As an example consider the case of an old software system where the only reliable documentation is the source code. The use of tools to automate partially the gathering of information from the listing of the source code and the conversion of this information into knowledge contrasts starkly with the case where reliable documentation exists and the information-gathering is mainly manual, that is, the maintainer reads the documentation to establish his conceptual model of the system.

Current descriptions of maintenance offer the highest-level view of maintenance, the *starting point* for a top-down refinement, culminating in the establishment of a more detailed

maintenance model. All the maintenance models surveyed were found to be compatible with the definition of maintenance given in chapter 1. The generalised model of the maintenance process can not be viewed in isolation however; the model implicitly admits of the existence of an organisation which will implement the model. The author envisages a maintenance organisation whose structure is a hierarchical one, consisting of three main levels:

1. Managerial level
2. Supervisory level
3. Technical level

Associated with each level of the organisation is a different type of information. The relationship existing between a particular level within the hierarchy of the maintenance organisation, and the type of information utilised by that level is shown below in Figure 2.1. The type of information that each level of the organisation utilises has important implications for objects stored in the object base of a maintenance support environment:

1. The range of granularity of the objects requires that appropriate tools are available to manipulate, store and retrieve these objects.
2. The types of information utilised by the organisation can be used to produce type classes, so enabling the 'typing' of objects in the object base, by means of an attribute which indicates the category of information stored in the object. In this way a tool is prevented from accessing an inappropriate object, or accessing an object in a way that is incompatible with its format and/or content.

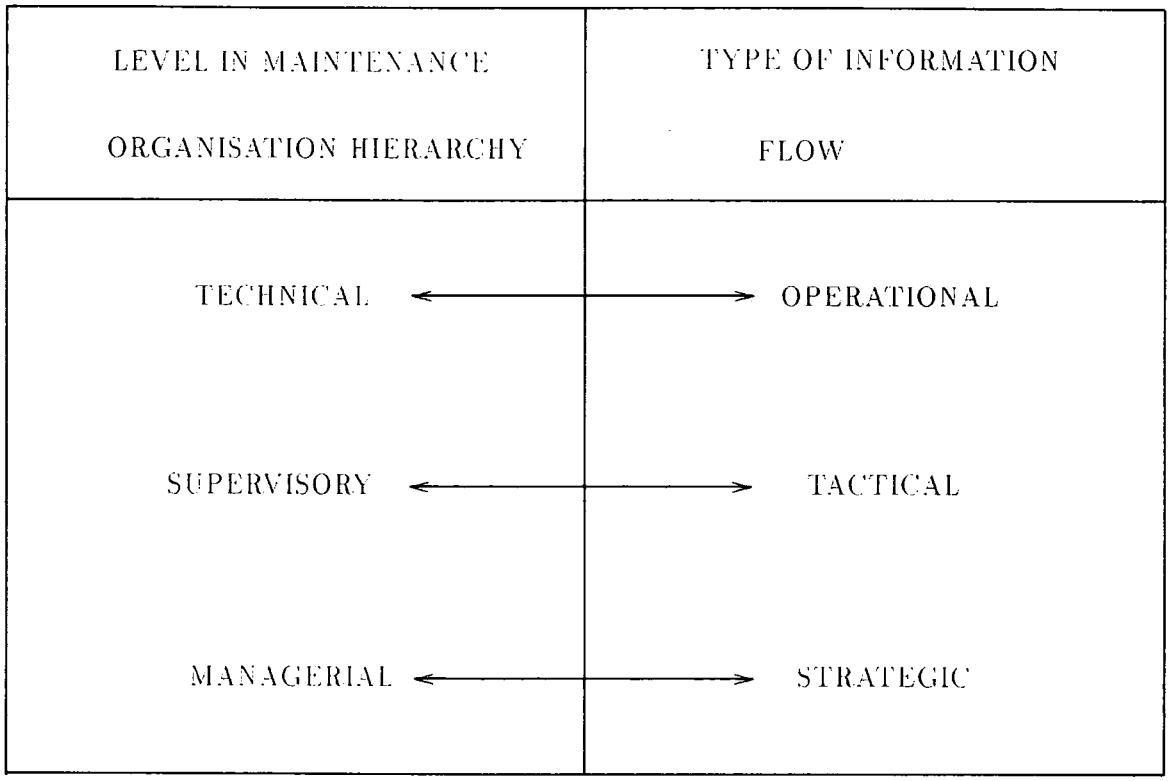


Figure 2.1: A Generic Maintenance Organisation Hierarchy and Associated Information Types

In addition, these types of information indicate the desired functionality of a software engineering object management system, discussed in chapter three. As an example of functionality, a highest level grouping in the object base is required, i.e. the partition level. This is needed, primarily for the operation of a management information system, regarding the tactical and strategic information used by a maintenance organisation. The purpose of such information is discussed below.

Information flows within each level and between levels are determined by the hierarchy of the maintenance organisation. There are two vital functions performed by management, i.e. communication and control, which depend on the capture, interpretation, utilisation and distribution of information. The managerial level of the maintenance organisation delegates to the supervisory level, which implements the long-term strategy of the organisation. The supervisory level also delegates to the technical level of the organisation those tasks necessary to fulfil the maintenance objectives determined by the managerial level; the information flows are bi-directional.

It is not possible to specify more precisely the nature of the hierarchy of a maintenance organisation since it is, of necessity, organisation-specific. This also applies to the model of the maintenance process, since it is subject to the constraints imposed by management philosophy; in addition the maintenance model is likely to be application-dependent. The type of information utilised by each level within the hierarchy of the maintenance organisation is outlined below.

1. Operational

This is the lowest level, and the information here is very detailed and is specific to the *function* of the maintenance organisation. Operational information may not even be communicated, if the person having the information does not feel that it needs to be. This information could be the understanding that a maintainer achieves of software but does not record, making the work of future maintainers more difficult. When operational information is communicated, it may not be actually documented but may instead be communicated orally, or by electronic mail when:

(a) The emphasis is on speed and accuracy, and the production of hard-copy may introduce inaccuracies.

(b) The quality of presentation of the information is not a critical factor.

This type of information is mainly concerned with the technical aspects of maintenance, and is concerned directly with software.

2. Tactical

The information at this level is used to monitor the resources, (men and machine), used in achieving the strategic objectives of the maintenance organisation and includes the application of access and financial control to large collections of information associated with maintenance projects. In the context of a maintenance organisation this information is concerned with the economics of maintenance, particularly with regard to the productivity of the organisation. This kind of information tends to be presented in the form of reports and summaries drawn up monthly or quarterly.

3. Strategic

This type of information is used by senior management for long-term planning and

its structure is not as predictable as that of tactical information, since a strategy is often formed in response to external factors, such as changing trends or new results from research initiatives. The information does not need to be highly-detailed or excessively accurate, since the long term forecasts produced at this level may be subject to distortions due to factors beyond the control of the organisation.

The generalised maintenance model adopted as a basis for the ISMSE is shown below. Its high-level view summarises the main activities undertaken during the course of a maintenance assignment; the model pertains mainly to the technical and managerial aspects of the maintenance process.

1. Verification of need to modify the software i.e. the program *and* its associated documentation
2. Understand the software
3. Modify the software, including the documentation
4. Validation of the software (i.e. the functional specifications) and regression testing

This high-level view or generalised model of the maintenance process should not be regarded as immutable, since research will cause the model of the maintenance process to evolve. Since, as mentioned earlier, the role of the environment must support the effective use of an effective process, it is apparent that the environment must be capable of evolving in response to the evolution of the maintenance model. As pointed out by Kaiser [61] the crucial test of any environment is whether it can support its *own* maintenance.

2.3.4 Discussion of the generalised maintenance model

Each of the categories from this generalised maintenance model are now discussed.

2.3.4.1 Problem verification

The earliest phase of the maintenance process is the receipt of a maintenance request (often from users); this may be an error-condition report or a request for the enhancement of the program. However a need for maintenance may arise through evolution of the maintenance organisation, and/or the evolution of the organisation, of which the maintenance organisation is a part.

Preliminary work is then undertaken to verify that maintenance is necessary. This preliminary work is greatly facilitated if there exists a database containing information regarding the history of the software system under scrutiny, which avoids repeating maintenance which has already been performed on a system. Typical information might include enhancements resulting in a new version of the software, or known 'bugs' and the corrections made to the software to remove these. If attention is not given to this vital area, it may mean that a maintenance department becomes overloaded through having to 're-invent the wheel.' Mellor [83] suggests that 'the approach to dealing with failure in a software product should be a hierarchical one, each level of the hierarchy acting as a filter for the level above and that simple queries due to known faults should be answered on the first level.' According to Mellor the highest level of the hierarchy is the design authority for the product, who devises repairs to code and incorporates changes into future releases of the product.

2.3.4.2 Understanding the program

As already mentioned a maintenance assignment contains an understanding phase, and the environment must provide support for this vital aspect of software maintenance. If the maintainer's understanding of the system is incomplete, then it is not possible for him to safely modify the software - (note that this implies safely modifying the documentation as well as the program.). There are two facets to the understanding of a program.

1. A local understanding of the program, where the changes are to made, specifically, *how* the program does what it does.

2. A more global understanding of the program so that the modifications do not adversely affect the other parts of the program. (It should be pointed out that a *total* understanding of the program is unrealistic).

When a maintainer seeks to achieve an understanding of a software system his first recourse is to the documentation of the system. If he finds that related parts of the documentation are not in agreement then he is forced to seek his understanding of the system from the only reliable documentation, i.e. the source code. The elements of a software configuration (other than the machine code representation of the software system) are together known as the documentation, when they are in a human-readable form. Examples of such elements are requirements, specifications, design, source code, test cases, test results. The older the software, the more likely that some of these elements are either missing or are unreliable, i.e. they do not reflect the current state of the system. This is of crucial importance since, as

mentioned earlier, the starting point for gaining an understanding of software begins with a reference to the documentation of the system.

2.3.4.3 Modification

Only the models due to Martin and Patkau give any detailed guidance in this area. None of the models explicitly address the problems associated with the integration of software, in the context of software modifications. When new software is written and is to be incorporated into an existing software system, the problem of integration of this software arises. Using a top-down strategy, or a bottom-up strategy, or a combination of the two, are the choices available to the maintenance programmer. Which strategy is adopted depends on the preference of the maintainer, and also depends on the testing strategy adopted, for example, when is interface testing carried out in relation to the other elements of software testing ?

Configuration management is concerned with the interrelation of software components, such as requirements, specification, source code, and documentation which are the products of the respective phases of the software life-cycle, as well as dealing with the traceability between the products of these phases. In a large software system, this is a major task. This topic is referred to in chapter 3 in the context of object management.

Version control is concerned with choosing the correct version of each component which is part of a particular software system. Failure to ensure this will almost certainly compromise the functional integrity of the system. In the context of medium-to-large software systems which have a lifespan measured in years, and which are periodically updated to reflect changes in the problem or application domain, users may seek changes to *any* of these

versions. Keeping track of which changes have been made to each version requires that a complete version history of the software be kept. This topic is referred to in chapter 3 in the context of object management.

2.3.4.4 Validation

Following a modification to a software system, when changes have been made to the source code, only one element of the documentation of the program usually remains a true representation: the listing of the source code. It is generally accepted that the documentation of programs is a much neglected activity.

Liu [76] emphasises the need to test the *remainder* of the documentation relating to the modification, so that redocumentation of the system following a modification preserves the compatibility and consistency of the elements of the software configuration.

Failure to test the whole of the documentation relating to the modification may mean that if this part of a software system is prone to modification (e.g. part of an accountancy application program which deals with tax thresholds,), failure to test the redocumentation may mean that this part of the program is redocumented incorrectly. A future maintainer, not regarding the documentation as reliable may redocument this part of the system and so 'layers' of documentation build up making the software larger, more complex and more confusing for any future maintainer. Unless the compatibility and consistency of the software configuration is preserved, following a modification, the end-result is increased complexity and therefore degradation of the system.

When a maintainer seeks to achieve an understanding of a software system his first recourse

is to the documentation of the system. If he finds that related parts of the documentation are not in agreement then he is forced to seek his understanding of the system from the only reliable documentation, i.e. the source code.

During the validation phase none of the maintenance models emphasises the course of action to be taken when there are elements of the software configuration missing, e.g. when there are no test cases or test results included in the documentation of an ageing software system. In the absence of test cases or test results it is impossible to carry out regression testing after a modification has been made to the source code and so the proposed modification will have to be postponed until test cases *have* been devised and test results obtained for these cases; only then can the program modification be said to be free from any 'ripple effect', i.e. the modification does not adversely affect the remainder of the software.

Only Carter [24] mentions any recovery from the result of the failure of the software, i.e. 'rollback' and file update.

2.3.5 Summary

A maintenance support environment must actively support the maintenance organisation using a generalised model of maintenance. The technical aspect of the maintenance model is something which depends on the context in which maintenance is being performed, and so is flexible, and can be refined to suit the type of maintenance being carried out, e.g. perfective change, adaptive change.

The maintenance models due to Boehm (1976) [15], Liu (1976) [76], Sharpley (1977) [116], Parikh (1982) [100] Yau (1982) [130], Carter (1986) [24], and Chapin (1988) [25] concentrate mainly on the technical aspects of maintenance. Martin (1983) [79], Patkau (1983) [102], give a more comprehensive description of the maintenance process.

In the context of the creation of a maintenance support environment it is insufficient to view the technical aspect of maintenance in isolation of other aspects of maintenance, such as the managerial and organisational aspects. To facilitate the evolution of the technical aspect of the maintenance model and the consequent evolution of the support environment, management must be able to access information concerning maintenance assignments quickly and easily. Hence the maintenance model must integrate all aspects of the maintenance process.

Chapter 3

Integrated Software Engineering Environments (ISE)

3.1 Introduction

An integrated software engineering environment is a generic term for a collection of software tools, available to the software engineer via a command language or a system of menus. The nucleus of such an integrated software engineering environment is a database or knowledge base which may also act as the interface between the tools. The meaning and importance of

the term 'integrated' is expanded upon below.

Ideally an ISE should be both language-independent and method-independent; its area of applicability being dictated by the tools which are integrated into it.

3.1.1 The advent of Integrated Software Engineering Environments

The perceived need for Integrated Software Engineering Environments (ISEs) has arisen because:

1. Software systems are becoming larger and more complex than was ever envisaged, owing to the evolution of ever more powerful computers, through advances in computer hardware. Integrated Software Environments were conceived as a means of improving the productivity and quality of software through the use of a fully-integrated compatible set of software tools, and incorporating modern software engineering techniques into software design and development, a spin-off benefit being the freeing of programmers for more creative activities.

2. ISEs are used for development of software and 60% [70] of software maintenance is enhancement - it is therefore apparent that an ISE offers support for the maintenance phase of the software life cycle. From the development standpoint, the development of software is being restricted by the maintenance backlog - increasing maintenance

productivity will release human and non-human resources for software development.

One of the first environments to provide *true* integration was CADES [81, 117]; many environments which claimed to be integrated were loose assemblages of tools, methods, and practices. The notion of environment integration is developed further in the remainder of this chapter.

3.2 Overview of Integrated Software Engineering Environments (ISE)

The important features of an environment are:

1. its architecture - because it is the implementation of the design of the environment.
2. its interfaces - because, as will be explained below, the integration of the environment takes place at these interfaces.

These features are described below.

3.2.1 Environment Architecture

Most publications concerning integrated environments give little indication of what is meant by the term 'integrated', but instead rely on a tacit understanding of the term; exceptions are Delisle [33] Houghton [55] and Lewerentz [69]. Delisle [32] views integration of an environment as:

> '...something which causes the environment to appear as a single tool to the user, so the user does not have the problem of performing mental context switches when using various functions within the environment.'

Houghton [55] defines integration as:

> 'The close unification of the major functions or processes of an environment ...'

and points out:

> ' ...that wherever an interface occurs in an environment, there is a need for integration.'

Lewerentz [69] defines integration as:

> '...the smooth interaction of tools, all tools having a uniform user interface.'

The IEEE Standard Glossary of Software Engineering Terminology [57] defines integration as:

> 'The process of combining software elements into an overall system.'

This definition has generality but requires amplification according to the context in which it is being used.

The author defines integration as 'The incorporation of software tools into a coherent unit for the generation and management of information concerning a software system.' The purpose served by an integration mechanism is to 'dovetail' the components of the environment so that they may work 'in concert'. One of the main features of an integrated software engineering environment is that information collected using one tool can be made use of by other tools.

Many software engineering environments claim to be integrated but this claimed integration of tools is disputed by some authors, e.g. Dart [31] and Hansen [53].

If the integration issues in Integrated Software Engineering environments are not clearly specified, it is not possible to see whether effective integration of the environment has been achieved; the concept of environment integration must be clearly and explicitly stated so that the architectural principles underlying the design of an integration mechanism can be established.

3.2.2 Environment interfaces

In order that the main interrelated components of the environment may be incorporated into a coherent unit, it is first necessary to identify the interfaces where these components meet. As an analogy, the integration of the interrelated components of a motor vehicle powered by an internal combustion engine serves to illustrate this assertion.

The chemical energy which supplies the motive power for the vehicle is locked up in the fuel as chemical energy. A trigger is necessary to release this energy and this is supplied by an electrical spark. An integration mechanism is necessary across the interface between the fuel and the electrical spark - the integration mechanism is the carburettor, which ensures vaporisation of the fuel and mixing of fuel vapour and air in the correct proportions prior to ignition.

The power generated by the energy in the fuel needs to be converted into motive power which can be applied to the wheels. Integration across this interface is achieved by means of a transmission system whose main components are the gearbox and clutch.

Each of these main components of the motive power system can be subdivided to reveal more interfaces with their accompanying integration mechanisms, e.g. the reciprocal motion of the pistons needs to be converted into rotary motion, so that the transmission system can impart motive power to the wheels. Integration is required between the engine and the transmission system: integration across this interface is facilitated by the crankshaft.

Since there are many interfaces in an environment, then there are many instances of integration, so that the architecture of an ISE is inextricably linked to the integration of its functions, notably through the integration of its main components, the toolset and the database. Houghton et al [55] observed that the integration of an environment requires the integration of at least three interfaces.

1. user interface
2. database interface (tools communicate through here)
3. machine interface

Houghton et al also showed [55] that the user interface and database interface are the most important interfaces, and that the interfaces can be viewed by modelling the system as a series of abstract levels, as shown in Figure 3.1.

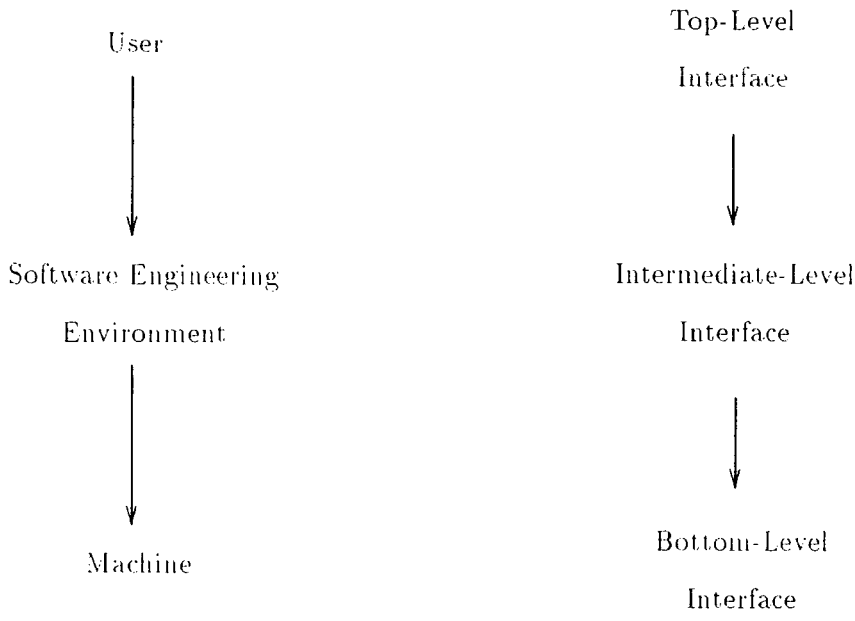


Figure 3.1: Modelling of Environment Interfaces

Many such environments have layered architectures which shield the user from the underlying operating system, and may even use *existing* environments as an intermediate level, for example Unix. Here the Unix interface may be used to invoke the Unix tools, the Unix tools communicating with the underlying Unix primitives. The architecture of a software engineering environment based on Unix is shown below in Figure 3.2.

3.2.2.1 User interface

An integrated software engineering environment supports the co-ordination and management of tools via a high-level user-interface. The usual type of user-interface is:

1. WIMPS (Windows, Icons, Pull-down Menus)

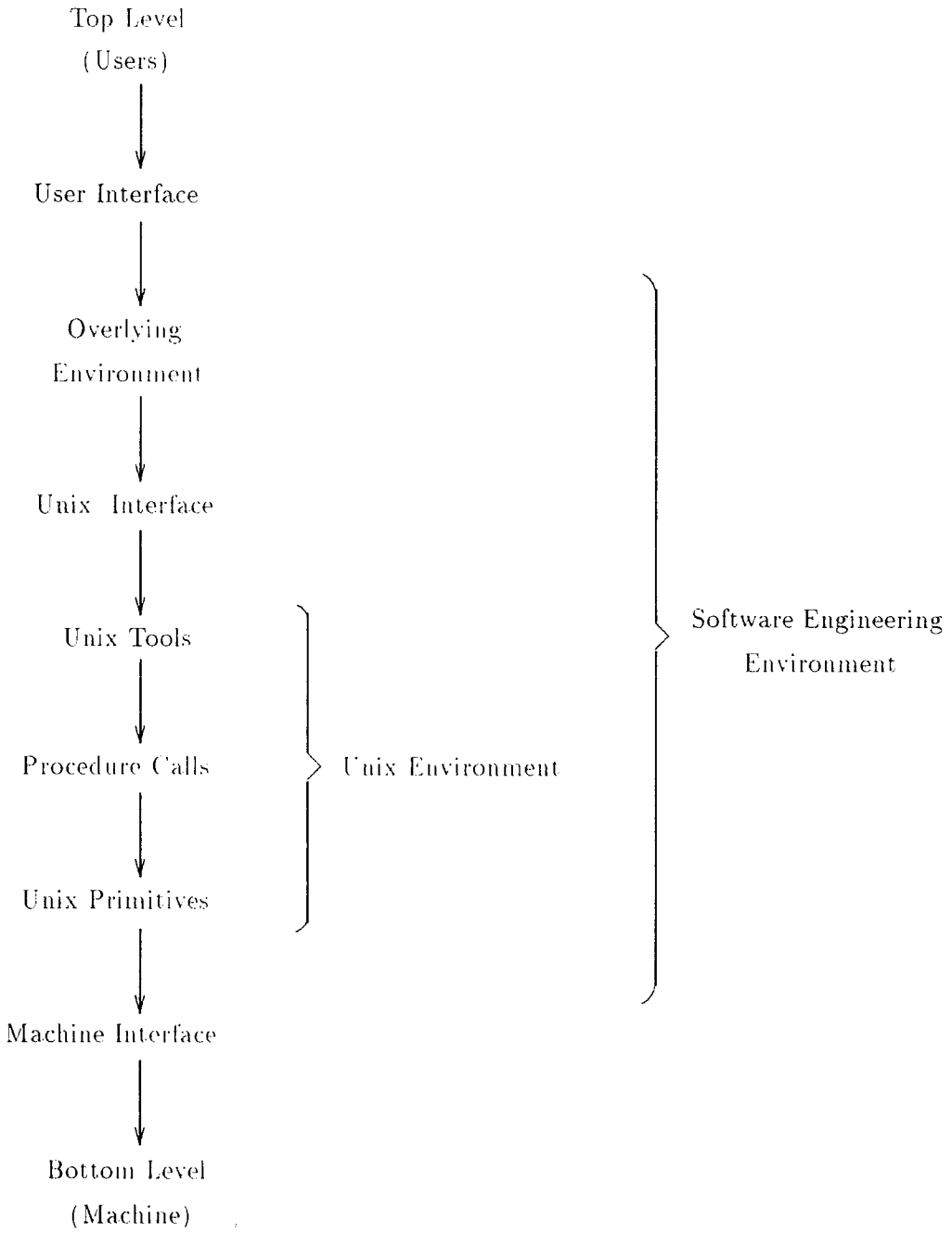


Figure 3.2: Generic Architecture of a Software Engineering Environment based on the Unix operating system

2. Command language

3.2.2.2 Tools interface

Integrated software engineering environments which claim to be extensible and tailorable to the users' requirements must be able to incorporate third party tools. The definition of a tools interface to allow the incorporation of 'third-party' tools must determine the degree of control a tool has over the operating system and database; and also take into account the facilities needed for an effective user-interface.

3.2.2.3 Database interface

The interface to the database may be environment-specific or may be from a DBMS package. Apart from being a repository for information, the database can be viewed as a tool that is used by other tools, and therefore acts as an interface between tools.

3.3 Classification of Integrated Software Engineering Environments

The two most complete attempts to classify ISEs are due to Dart et al [31] and Houghton et al [55].

3.3.1 Dart Classification

Dart et al [31] classify ISEs as:

1. Language-centred.
2. Structure-oriented.
3. Toolkit.
4. Method-based.

3.3.1.1 Language-centred environments

These are built to support the coding phase of a particular language, whose toolset is tailored to that language. These environments are highly-interactive and offer limited support for large scale programming efforts. Their most important features are:

1. They offer support for prototyping, since the development environment and the run-time environment are the same. Small changes to code can be made executable very quickly, allowing programs to be incrementally built.
2. Semantic information from symbol tables is recorded by the environment and is available to the programmer via tools such as browsers. In this way the programmer may achieve a deeper understanding of the software under construction or when the programmer is engaged in maintenance.

3. Support is provided for configuration management and version control, but no support is provided for project management.

3.3.1.2 Structure-oriented environments

These environments support the coding phase of a particular language; the user deals *directly* with program constructs, thus avoiding the tedium of having to remember details of the syntax, enabling the manipulation of program structures in a language–independent manner.

Semantic information can be attached to program structures and made available to the user. Structure-oriented environments have as their main component a syntax-directed editor through which all structures are manipulated, and is also the interface through which the user interacts with the environment.

Programs can be viewed at different levels of abstraction and detail, multiple views of programs can be generated from the program structure, and browsing of these views is supported through the use of windows.

Structure-oriented environments have the ability to formally describe the syntax and static semantics of a language from which an instance of a structure editor can be generated. The structure editor reports syntax and static semantic errors as soon as they arise, making possible the incremental checking of semantics. The user interacts directly with program constructs and avoids the tedium or remembering details of the syntax.

Generation of structure-oriented environments is made possible through encapsulation of the syntactic and semantic properties of a language in a grammar.

3.3.1.3 Toolkit environments

These environments are language-independent and mainly support the coding phase of the software development life-cycle. They consist of a collection of small tools which are not controlled or managed by the environment. Facilities are provided for version control and configuration management, but little support is provided in terms of consistently and automatically managing user activities, since the toolset is only loosely coupled to the environment.

A high-level interface needs to be placed on top of the normal user-command interface, thus increasing control over tool usage. Toolkit environments such as Unix, allow the user a high degree of tailoring, but such environments provide little in the way of environment-defined management or control techniques for using the toolkit. In addition, little support is provided for the maintenance of large software systems.

3.3.1.4 Method-based environments

Support is provided for a broad range of software development activities, such as team and project management. Tools for particular specification and design methods are incorporated into the environment. These environments either support:

1. A particular development method.

   The development method may include any of the following:

   (a) Specifications

   (b) Design

   (c) Verification and validation .

   (d) Re-use.

   Different methods exhibit different degrees of formality, i.e. informal, (text), semi-formal (textual and graphical descriptions with limited checking facilities), formal (with an underlying theoretical model against which a description can be verified). Examples of formal methods for specification are Petri nets, state machines and specification languages.

2. Methods for managing the development process.

   Facilities are provided for version, configuration and release management along with procedures and standards for performing these tasks consistently.

3.3.2 Houghton Classification

The classification of ISEs by Houghton et al [55] is related to the phase(s) of the software life-cycle supported. The classes are:

1. Programming.

2. Framing.

3. General.

3.3.2.1 Programming environments

These provide support for the coding, debugging and testing of programs, which are written using a high-level programming language.

3.3.2.2 Framing environments

These concentrate on the earlier phases of the software life–cycle, where the system is *framed* by its requirements and design. Implicit looping in the life-cycle means that all phases of the life-cycle are supported: for example changes in requirements resulting from errors detected during operation of the software means that the operation and maintenance phases are being supported, though this does not include all activities within each of these phases.

3.3.2.3 General environments

These environments contain basic tools which support all phases of the software life–cycle, and usually support more than one programming language. They often contain advanced special–purpose tools, for certain phases and can be adapted to most methodologies. An example of a general type of ISE is the IPSE, whose impact on software engineering in recent

years has been considerable.

3.3.3 Comparison of Dart et al's and Houghton et al's classifications

Dart et al's classification is deeper and more detailed, taking Houghton's classification of programming environments further by breaking it down into Language-centred, Structure-oriented, Toolkit, and Method-based environments. Houghton's framing environments is included in Dart's method-based environments, and Houghton's general environments is equivalent to Dart's toolkit environment.

3.3.4 The European Alvey Integrated Project Support Environment (IPSE)

The forerunner of the IPSE was the Ada Programming Support Environment (APSE), many features of the APSE are present in the IPSE. It is useful, therefore, to examine briefly the architecture of the APSE and some of its functions. The Stoneman document [23] summarises the objectives of the APSE, the main objective being the provision of cost-effective support to all functions in a project team engaged in the development, maintenance and management of a software project.

The APSE is comparable with early generations of IPSEs; since it only supports the programming phase, it can be thought of as a sub-set of an IPSE. its architecture is shown below in Figure 3.3.

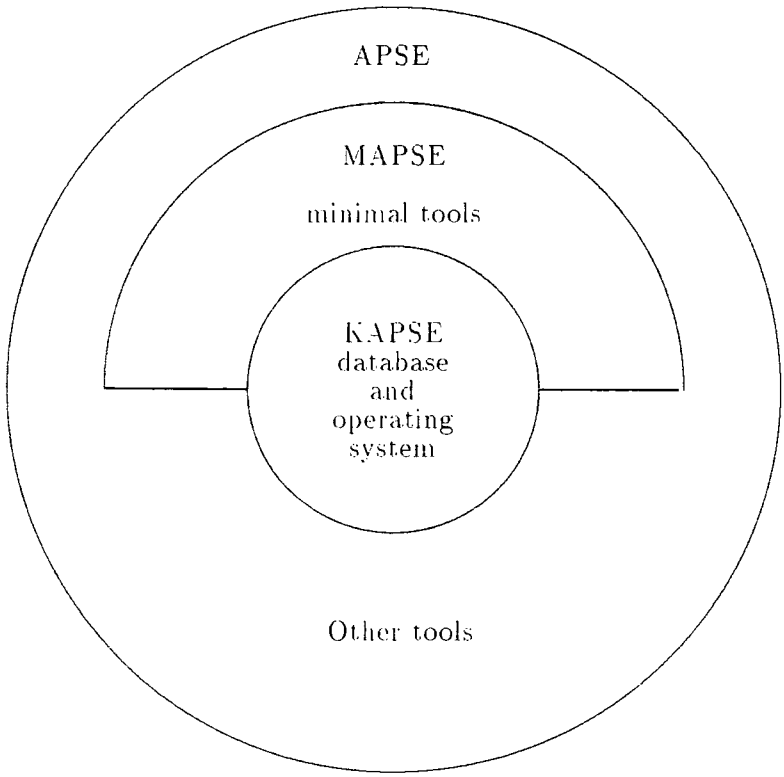


Figure 3.3: The Architecture of the Ada Programming Support Environment (APSE)

The APSE is a coordinated set of tools around a common database. The central item is a kernel presenting an interface to the tools outside. The kernel acts as a bridge between the tools and the operating system, it is implemented on top of the host machine. The interface is host-independent, and since the tools, compiler, editor, etc., operate on top of this interface they are also host independent, i.e. the complete tool set is portable. To move it to a new host, only a new kernel is needed.

The single most important feature of the kernel is its database - it is through this database

and its built-in constraints and structures that a consistent and reliable set of interfaces can be presented to the tool builder and user - this particularly applies to recovery mechanisms where individual tools cannot have adequate information to cope with situations outside their control. The database is typed, i.e. it has knowledge of certain properties of the objects that it contains, and can prevent their misuse by incorrect tools; this simplifies tool development by providing the tool writer with an appropriate framework.

3.3.4.1 Overall objective of the IPSE

An IPSE has been defined [78] as: 'An integrated compatible set of tools based on a methodology for all parts of system development and operation, sharing a common database.'

As pointed out by Stenning [120], the software industry is essentially a manufacturing industry. Manufacturing industries have much experience and knowledge gained in tackling large-scale engineering projects, and rely heavily on tools and techniques which are the fruits of disciplines such as operational research. Building a large software system is essentially a large-scale engineering project; this philosophy was adopted by the Alvey directorate. Three generations of IPSE were envisaged, leading to the establishment of the Information Systems Factory (ISF). The goal of the ISF is the provision of a set of tools for producing IT systems on a 'one-off' basis, using sound software engineering techniques.

In outline, an IPSE is developed by integrating software engineering tools into a common database structure, that is, the tools share a database, which is the nucleus of the environment. This is a disciplined engineering approach, enabling project management information

to be extracted easily and efficiently, making effective project management and support possible; the aim is for the IPSE to be language-independent and method-independent, the tools which are integrated into the IPSE dictating its area of applicability.

3.3.4.2 Evolution of the IPSE

The main distinction between each generation is the level of integration achieved for the various tools within the IPSE. Integration means that information collected using one tool can be made use of by other tools, that is, tools know about each other. Mair [78] summarises this evolution as occurring in three generations.

First-generation

A typical first-generation IPSE has a Unix-like file-based database combining off-the-shelf management and software support tools, which are normally presented to the user by means of an enhanced user-interface, rather than the cruder Unix command interface. The tools are incorporated with little, if any, modification, and the tool is used within the IPSE as it would be on its own. Invocation of the tool is usually controlled by the IPSE, using standard Unix call procedures. Any read/write files used by the tool are generally under the control of some form of configuration manager built into the architecture of the IPSE. The structures of the files themselves remain unaltered from the stand-alone version of the tool but form part of the overall filebase.

Second-generation

Some second-generation IPSEs are commercially-available now. These are closed IPSEs. They have a relatively limited tool set, offer no tool interfaces, and therefore cannot host 'third-party' tools. This means that they are in danger of becoming obsolete, through being unable to keep up with advances in tool design. These IPSEs have more of a true database structure, rather than a simple filebase. Entity-relational database models are used to hold the information of the IPSE. The primary element is the entity, an object within the database; each entity can have attributes assigned to it - these are its properties. This aspect of an IPSE is discussed in more detail, later in this chapter.

Second-generation IPSEs integrate tools at a much lower level than simply holding them in a database as an isolated component. Instead tools are held in the database as entities, having traceable links to their files, also held as entities. All information relating to the operation of the IPSE, as well as the data produced by it, resides in the database as different types of entities.

Because of the level of the integration required by second-generation IPSEs, many of the tools present will have been purposely developed for that IPSE; a large number of these purpose-built tools are due to the result of research undertaken specifically for IPSE development. It is envisaged that tools will be used serially, when one tool has completed its task, the next tool is invoked.

Third-generation

Integration will extend the database concept into a knowledge-based system. Instead of using

tools serially, fully-integrated tool sets will allow the user to freely interchange between one task and another. The success or failure of an IPSE depends on its ability to incorporate 'third-party' tools into its structure, i.e. its architecture must be open.

3.4 The Suitability of an IPSE as an Integrated Software Maintenance Support Environment

3.4.1 Introduction

It has been pointed out [55] that it is infeasible for any one environment to provide *complete* support for *all* software engineering activities that may be carried out during one revolution of the software life-cycle: however, Integrated Project Support Environments (IPSE) are the most complete attempt to achieve precisely this.

The role of an IPSE is to provide support for the software engineering process. Software Engineering is defined by IEEE [57] as 'The systematic approach to the development, operation, maintenance, and retirement of software.' It is apparent that if an ISE is able to provide support for these activities, it must be able to provide support for the respective underlying process models. Current IPSEs are ISEs whose intended goal is the provision of *fully*-integrated support for the software process.

Glass [46] is of the opinion that software maintenance can be regarded as the entire range of software development, in microcosm. This notion should be tempered with caution since software maintenance differs from software development in one very important respect, in that it must be performed within the constraints of *existing* software, whereas a software developer begins with a 'clean sheet' when developing a new software system.

3.4.2 IPSE support for maintenance

The suitability of an IPSE for providing assistance for the maintenance of software, and in improving the productivity of a maintenance organisation can be illustrated by reference to the two main features of an IPSE, i.e. object management and tool integration.

3.4.2.1 Object management

The database is the unifying element in the architecture of an IPSE and also acts as a central repository for all information associated with a project. The advantage of using a database, instead of a file based storage system, is that centralised control can be exercised over the data, enabling safeguards to be employed in recording, querying, and manipulating the data. Implicitly, this means that a higher degree of data integrity is possible than can be obtained when using a simple file-based system as the basis for a persistent data store.

IPSE databases store objects: an object is a name given to a collection of information which

has a unique identity. An object has attributes, which describe the nature of the object: the list of possible attributes is open-ended, but those attributes which record history, categorization and access rights are essential for the *management* of such objects. For this reason the database is often referred to as an object base, and the term Database Management System (DBMS) is replaced by Object Management System (OMS). Examples of objects are natural language text, source code, requirements specification, test data and configurations; such objects are large-grained and require the use of large-grained tools. This approach contrasts with, e.g. toolkit environments such as Unix, whose tools communicate through 'pipes', which enable single files of textual information, to be passed from one tool to another. This view of the complexity of the data objects and structures which tools must, of necessity, create and communicate to each other in an IPSE is inadequate.

The concept of an object has great importance for the maintenance of software as regards:

1. **Understanding of software**

   Linked with the notion of an object is the notion of a view, which has important implications for the understanding of software. A view is a data management term defined [57] as 'A cross-section of a software system which contains objects relevant to a particular task.' It should be noted that a view can also be a *single* object, e.g. a configuration. Other examples of views are versions, call-graphs. Understanding of a program is achieved by examining different *views* of a program, no one view being sufficient to permit this understanding. Some views of a program come readily to hand, e.g. a linear view of a program is represented by the program text, but a hierarchical view of the program requires the use of a tool to produce, for example, the call-graph

structure. Similarly, other views of a program, such as the statements referencing a variable or the procedures using a particular module require the use of appropriate tools.

2. **Version control**

A version is defined [57] as 'The latest instantiation of a software system which has superseded all other instantiations.' A later version of a software system may have been produced to correct errors in, or to add enhancements to, an earlier version. Control of versions makes possible their correct use, possibly by restriction of access to existing versions, and the creation of new versions. The attributes possessed by an object in a database make it uniquely identifiable: a *group* of related versions, each of which meets some specified criteria can be regarded as different versions of the same abstract-object. Within such a group a particular version can be assigned to be the default version, which can be useful in reducing access times.

3. **Configuration management and control**

A software configuration is defined as a collection of software elements or objects, (also known as a configuration item), that performs some well-defined function, e.g. the modules which together constitute a computer program.

A software configuration can itself be regarded as an object. The aim of configuration management is to ensure, that a software configuration is properly constituted to perform its function throughout its lifetime, through the selection of the appropriate version of each individual component, new versions of components being the result of maintenance activity. Configuration management is sometimes performed as an integral part of the software development process, retaining control over the evolving

software, sometimes it is a discrete activity, being triggered by each new revision of the software.

A software configuration may have a long lifespan, e.g. a new release of a software system or a short lifespan, e.g. a system prototype, produced during the development of an enhancement to software. Software configurations are themselves objects and may therefore exist in version groups, the relationships between such objects are often very complex, and may be related in time, such as consecutive releases; others may co-exist in time, e.g. separate prototype models. An object management system must concern itself with the generation, release and subsequent control of configurations, and must be able to furnish details concerning the components of any configuration as regards their history and antecedents, through the use of history attributes. In particular, in-built constraints of the object management system as regards its operations are necessary to preserve the validity of an object's history attributes, e.g. which compiler option was used to produce the object file.

3.4.2.2 Tool integration

In general software tools increase the productivity and power of the software engineer. In the context of software maintenance they remove much of the drudgery of searching for information, but can provide an embarrassment of riches, i.e. they can provide *too much* information. Tool integration in an IPSE means that:

1. Tools can communicate with one another, permitting abstraction of the information at the desired level of detail. The adoption of such a strategy for making use of automation, in conjunction with an object management system to provide views of the software provides a powerful tool for maintenance.

2. The ability to incorporate 'third-party' tools from tool vendors, using the tools interface provided by an IPSE means that a maintenance organisation can keep up with the latest advances in software tool technology.

3.4.3 Problems associated with IPSEs

1. They consume large amounts of machine resources.

2. The definition of a tools interface to allow the incorporation of 'third- party' tools, i.e. tool integration, which must determine the degree of control a tool has over the operating system and database; and also take into account the facilities needed for an effective user-interface. The arguments centre on the *complexity* of a tools interface, there being two main issues:

   (a) The degree of control a specific tool has over the operating system and database.

   (b) The facilities provided for an effective tools interface.

No international standard for interfacing tools to an IPSE has been specified but there are two contenders, PCTE [78] and CAIS [78]. The standardisation is important for two reasons:

1. Tool developers either align themselves with one IPSE or produce several versions for different IPSEs.

2. From a user's standpoint, will choosing an IPSE tie them to that system for years, causing them to miss out on new developments ?

3.5 Summary

The reasons for the advent of integrated software engineering environments have been given, and the meaning and importance of environment integration have been described in terms of environment interfaces. A review of initiatives in the classification of software engineering environment *types* has been made and an overview of available software engineering environment types has been given.

A description of the classification of software engineering environments has been given, together with the reasons for their advent. The architecture of a generic software engineering environment has been described.

The evolving nature of the European Alvey IPSE, with respect to its architecture, has been described; particularly with regard to its public tools interface (PTI), and the influence of the level of tool integration on productity gains.

The suitability of the IPSE as a maintenance environment has been outlined in terms of

its main features, its object management system (OMS) and its public tools interface, and the aspects of maintenance supported by them, i.e. program understanding, version control, and configuration management and control. The problems associated with IPSEs have been briefly described.

Chapter 4

Literature Survey of Current ISEs

4.1 Introduction

To improve the productivity of the maintenance organisation, automated support is required for the maintenance of large software systems. To achieve this end a software engineering environment must be able to provide facilities to support the technical, managerial and organisational aspects of software engineering, including re-use, over the complete software life-cycle, from requirements definition to maintenance. In particular, support must be offered for the *management* of the maintenance process, exerting control over the maintenance

process and enabling the accurate monitoring of its progress. It must be possible to extract information from that system, showing the state of that system, at any time.

In chapter two a generalised maintenance model was proposed: in this chapter a literature survey of software engineering environments is undertaken. The purpose of this survey is to determine whether existing integrated software engineering environments contain features that could be useful in an ISMSE: the results of this literature survey form the basis for the high-level design for a maintenance support environment, which is the subject of chapter 6.

4.2 The Survey

Introduction

Environments which only provide support for small subsets of the maintenance process, such as version management, have not been included in the survey. Even though the Software Engineering Environments surveyed have widely different objectives, they satisfied at least one of two important criteria, each of which, in isolation, would provide support for the keystone of software maintenance, which is program understanding. These criteria are:

1. They contain an integrated toolset, to aid in information capture
2. They have the capability to *manage* information, supporting abstraction, views, and the creation of information structures

The approach adopted in this literature survey was:

1. To ascertain the compatibility of those software engineering environments which *explicitly* support the maintenance process, with the generalised maintenance model proposed in chapter 2.

2. To evaluate other integrated software engineering environments, to ascertain the support they *implicitly* provide for the maintenance process. This part of the survey divides Integrated Software Engineering Environments into:

   (a) Non-hypertext environments

   (b) Hypertext environments

In the following literature survey the environments were evaluated under the following headings.

1. Objectives

2. Architecture

3. Functionality

4.2.1 Environments providing *explicit* support for maintenance

4.2.1.1 Microscope

Objectives

Microscope [4] aims to help programmers understand and modify complex programs, providing support for evolutionary development, and the means to estimate the effects of a proposed change, i.e. the ripple effect. Microscope's aim is to provide the programmer with the view of the program that the programmer wants.

Architecture

Microscope has a layered architecture, built on top of the host operating system. The knowledge base and user interface are shared by the tools.

Functionality

Microscope is a knowledge-based programming environment that includes tools to statically analyse source code, storing the results in the knowledge base, providing support for abstraction, and for obtaining views of a program. It is language specific, targeting CommonLisp and CommonObjects. Microscope is windows-based, each window having items that have annotations associated with them: an annotation is a piece of related program information in *addition* to documentation. Items may be nodes in a graph, symbols in code, stationary menus, or words or a phrase in a document. Associated annotations may include:

1. documentation

2. source code

3. constraints

4. defects

5. external view

6. revision history

An annotations menu lists links between items, for example, in a graph of a program's module structure, each node may be an active region representing a module, and have an associated menu, for example, the data-flow and control-flow between the two connected modules.

Microscope can display a program with any desired annotations: small amounts of information may be displayed directly in the program browser, for example, the number of times a function is called and its execution time can be displayed next to each node name. Other annotations, such as source code which are too large to fit in the program browser, are displayed in separate windows. Microscope supports a special class of annotations, called constraints, which are records of implied relationships between different parts of the program.

Dynamic Analysis

Microscope offers support for understanding by allowing the programmer to monitor execution, displaying changes in data structures and control flow dynamically, also saving the execution history. A programmer can become overloaded by too much information so, for example, Microscope allows the execution history to be analysed and filtered, providing the means for abstraction. The cause of run-time errors can be ascertained by examination of

state information and execution history, using flow analysis and the nature of the error to narrow down the possibilities. Monitoring-requests can also be made by the programmer, specifying for example, which events to look for and what subsequent action should be taken. Microscope can monitor the values held in a variable, or check the logic flow in conditional statements, record all function calls defined in a module, or provide program slices, for example recording all the loops that use a particular variable in their exit tests.

User interface

The user-interface is a graphical one to show the structural view of a program.

4.2.1.2 Arizona State University (ASU) Practical Software Maintenance Environment

Objectives

The environment [29] aims to support:

1. understanding software
2. changing software
3. tracing ripple effect
4. retesting changed software
5. documenting acquired knowledge
6. planning and scheduling maintenance tasks

Architecture

The tools share a database, no other information is given.

Functionality

The environment is language-specific, it operates on Pascal code which has compiled free of errors. The components of the environment include the personnel, the maintenance tools, and the software syntactic and semantic databases. The existing environment provides facilities to understand code, document code, and analyze code for ripple effects.

4.2.1.3 University of Colorado, Boulder - (prototype environment)

Objectives

The environment [99] is to be capable of supporting the interpretation of explicit maintenance processes, by coordinating the efforts of tools and personnel.

Architecture

No information is given except that the environment is to be open, to incorporate third-party tools.

Functionality

A common framework is suggested for understanding maintenance *and* the structure of an environment for supporting maintenance. The requirements for the environment include that:

1. the environment should be flexible

2. the environment should incorporate explicit process representations

3. users can alter tools and the process itself, as needed

The environment is not yet operational, but the support for maintenance provided by the environment is achieved by designing the environment around the notion of process programming. Process programming has, as its application, the domain of software engineering. A process program is defined as 'The static description of how a process could be carried out, incorporating the appropriate and necessary tools and object base.' Process programming was developed by Osterweil [99], its aim being to support the construction of a *family* of environments, each with its own view of the appropriate process model. In particular, the intent is to produce a software development environment kernel that can be parameterised by a process program; the environment is used in different modes according to which type of maintenance is to be performed on an applications program. The analogy of a process program and its effect on an environment is the cassette recorder - the recorder functions using the same set of components every time but the output depends on what is on the cassette - the cassette parameterises the program which operates inside the cassette recorder. Process programming regards the process model as malleable, i.e. it is *soft*ware and should be capable of adapting to changing circumstances. The notion of process programming is amplified in the discussion at the end of this chapter.

4.2.1.4 Genesis

Objectives

The main objective of Genesis [107] is to provide facilities to improve the productivity of

software developers, particularly in the evolution phase of the software life-cycle. Genesis is rule-based and particularly aims to support the evolution phase through its resource manager, using software libraries and version control.

Architecture

No information is provided concerning the architecture of the environment.

Functionality

The main components of Genesis are:

1. Resource manager (also called the Evolution Support Environment (ESE))

   This provides version control, traceability between software resources and methods for accessing these resources.

2. Activity manager

   This regards a project as successive invocations of software tools to manipulate a software resource. A rule-based protection mechanism provides the means for:

   (a) Pointing out potential inconsistencies

   (b) Automating certain activities for the user

   (c) Ensuring protection

   (d) Defining the software development methodology

3. Information abstractor

   This extracts relational information among the software entities of programs, stores the information in a database and makes it available to users via a high-level access utility, that displays the information in a form that can be easily understood.

4. Metric-guided methodology

   This indicates the complexity of the software at each phase, and suggests ways to reduce the complexity at that phase, as well as how to proceed to the next phase, so as to achieve the desired goals for the project.

4.2.1.5 United States of America General Service Administration's 'Programmers' Work Bench.' (PWB)

Objectives

The aim of the PWB [49] is to guide Cobol programmers through maintenance, testing, conversion and other activities. The PWB framework is intended to enhance the productivity of every member of the software management team, this software engineering environment is intended for the commercial sector.

Architecture

The workbench infrastructure is a framework for integrating software tools and controlling access to them. There is no *enforced* linkage between the tools, the environment is, in effect, a loosely-coupled toolset. Its structures include:

1. Architecture and systems management
2. Tutorials and information interchange
3. Reusable pattern program generation
4. Automatic JCL generation

5. Change control tracking

The main feature of this open-ended software management architecture is a table-driven sequence of screens to guide the programmer through specific development, or maintenance tasks, using specific tools.

Functionality

The series of menu-driven workbenches enables the user to customise his own workbench and toolset environments, including the addition of existing tools, while at the same time providing access to all other products in the PWB which provides:

1. The mechanism for enforcing installation standards and procedures, at the same time ensuring the integrity of the installation's software engineering environment.

2. The means for the setting-up of a library of reusable COBOL applications.

The PWB framework can be configured to include tools and facilities for operation in a database environment.

4.2.2 Environments providing implicit support for the maintenance process

4.2.3 Non-hypertext environments

4.2.3.1 Marvel

Objectives

Marvel [61] aims to support two aspects of an intelligent assistant, insight and opportunistic processing.

Architecture

No detailed information is given concerning the architecture.

Functionality

Marvel is able to:

1. provide a fileless environment to its users,
2. answer queries
3. coordinate the activities of multiple programmers and
4. automatically invoke tools.

The knowledge of the assistant is described in a model, and intelligence is achieved by interpreting the model, providing *insight* into a system and actively participating in development

through *opportunistic processing*. The concepts of insight and opportunistic processing are briefly discussed below.

Insight

The environment is able to anticipate the consequences of the user's activities, based on an understanding of the development process and the resulting software. The means that individual programmers can grasp more readily the structure and relationships in the software product, permitting a deeper understanding of their tasks. The environment is able to guide the programmer in changing a system, returning it to a consistent state, and also to help coordinate the activities of multiple programmers, enabling them to work individually, yet co-operatively, dovetailing their efforts.

Opportunistic processing

This means the automation of simple development activities, such as monitoring changes to the source code, invoking the compiler, and recording compilation errors, triggered by the user's action.

Components of Marvel

The main components are:

1. Object base

   This stores data as objects, in the object-oriented sense, the object base maintaining all the entities that are part of the evolving system, for example, information about the history and the status of the project, and the tools used in its development and maintenance. The object base defines the object classes and the relationships among objects, for example, one object is a component of another and when applied to another

object will produce a third. The object base is active, and accessing objects may trigger action.

2. Process model

   The model of the development process imposes a structure on programming activities: it is an extensible collection of rules specifying the conditions existing for the application of particular tools to particular objects. Some rules apply only when a user invokes a tool, others apply when the environment initiates tool processing and others apply to both cases. All the intelligence is encapsulated in the environment, instead of in individual tools.

Summary

The model embodied in the Marvel environment formalises the concepts of insight and opportunistic processing, which are two aspects of an intelligent assistant, by:

1. Maintaining all knowledge about both the specific development effort and the general development process in the object base.
2. Making multiple views of the object base available both to users and tools.
3. Modelling the development process as rules that define the pre-conditions and post conditions of development activities.
4. Gathering collections of rules into strategies.

The above rules allow Marvel to provide software engineering environments that intelligently assist development and maintenance efforts, by individuals, and by teams of users through

controlled automation, using available development tools. Marvel does not include any mention of tools for maintenance or of any assistance for the understanding of *existing* programs.

4.2.3.2 Aspect

This IPSE [52] is a collaborative venture between Systems Designers, ICL, MARI and the Universities of Newcastle and York.

Objectives

There are two key objectives.

1. An open environment allowing integration of third-party tools
2. The provision of a truly integrated set of tools sharing a common database structure, presenting the user with a consistent and coherent working environment

Architecture

Aspect is a distributed IPSE and has a layered architecture, built on top of the host operating system, which is a distributed Unix system. The public tools interface integrates the tools with the object base.

Functionality

Aspect allows a project to be divided among small teams of programmers via a hierarchical

directory system: these teams can work independently to develop parts of the system, using configuration control mechanisms to assist in the building of these sub-systems. Aspect is a multi-language distributed-host, distributed-target IPSE which aims to support all phases of the Software Development Life Cycle (SLDC), from specification through design and implementation, to testing and maintenance. It will also support project management, planning and control. The prototype was designed to demonstrate the feasibility and practicality of using a fully-integrated environment for real-time development, i.e. embedded systems. Aspect is being developed as an IPSE framework into which tools can be integrated, a minimal toolset is included for program development (Ada, Pascal and C compilers, and linkers.)

4.2.3.3 Eclipse

This IPSE [56] is a collaborative venture between Software Sciences, CAP, Learmouth and Birchett Management Systems, Lancaster and Strathclyde Universities and University College of Wales, Aberystwyth.

Objectives

Eclipse is an engineering prototype, a vehicle for trying out ideas and assessing their usefulness. The aim is to demonstrate the IPSE concept by building a practical system within the constraints of time scales and budgets, and to provide support for large-scale long-term projects, possibly geographically dispersed.

Architecture

Eclipse is a distributed IPSE, the central host computer is connected to workstations over an Ethernet Local Area Network, the host computer holding the central IPSE object base. The

public tools interface integrates the tools with the object base and the IPSE has a layered architecture, built on top of the host operating system.

Functionality

The process of system development is regarded as a series of transformations from one representation to another, with each stage introducing greater precision and rigour than the preceding one. Eclipse is a distributed IPSE but the tools are designed to run on the workstation itself. The nucleus of an IPSE is the database; objects in the database are produced by executing transformations (themselves stored as objects), which are defined in terms of UNIX commands. Problems encountered [78] with ECLIPSE include:

1. Complexity and control
2. Communications and management
3. Communications between computers

4.2.4 Hypertext Environments

Before surveying hypertext environments to assess their compatibility for software maintenance it is informative to examine the role of hypertext in software engineering.

The term 'hypertext' was first coined by Nelson [91] but the original concept is due to Bush [22]. Hypertext has been defined by several authors, for example [95] as non-sequential reading and writing. A good introduction to hypertext is provided by Conklin [30]. The main areas of software engineering in which hypertext can offer assistance are:

1. Program understanding

2. Information management

3. Abstraction

4. Documentation

The idea of using a hypertext system in conjunction with software engineering tools to form an integrated software engineering environment has been ascribed to Henderson [54]; the obvious advantage of this combination is that software tools can introduce automation to the extraction of information from the system and hypertext can provide the means of storing and retrieving this information, as well as allowing the maintainer to *create* information structures through the linking of objects in the database. The utility of hypertext systems for information management in large-scale software engineering through the diverse types of information permitted in hypertext nodes, e.g. text and graphics, has been demonstrated by Biggerstaff [13]

When the maintainer is presented with information from various sources he is faced with the task of organising this information so that he may be able to:

1. Use his reasoning powers to process the information into *knowledge*, thus gaining an understanding of the software, so that it may be safely modified.

2. Update the documentation of the software in order to make future modifications easier.

Hypertext offers support for information structuring, enabling the aggregation of objects, produced by tools, into structures which may have hierarchical and non-hierarchical organisations, which are derived views of a software system.

The following brief exposition based on an example from [11] is intended to serve as an illustration of the part that hypertext plays in relating information contained in the database of a software engineering environment. The basic components of a hypertext database are nodes, links and contexts. The nodes are a means of storing data and information and the links between the nodes are forged using *relationships* between nodes, the implementation of these links is via pointers in the database. Items of information describing a software system entered by the maintainer or produced by a software tool may be stored in the nodes of a hypertext database. The nodes containing information about the software system are linked to form a directed graph, known as the hyperdocument. Within the graph, the concept of contexts is used to partition [11] the data within the graph, providing support for configurations and version trees. Contexts, nodes and links all have attributes, and can assume values of strings, integers, reals or user-defined types. The function of attributes is to label the types of nodes, links and contexts. The attributes and their values are known as attribute/value pairs and effectively *classify* nodes, links and contexts.

A simple example of a link relating two nodes appears below, in Figure 4.1. The context has the attribute of 'software component' and its value is 'source code'; node 1 has the attribute 'module' and its value is 'initialisation'; node 2 has the attribute 'paragraph' and its value is 'input-routine'. The link has the attribute 'relation' and its value is 'uses'. The arrowhead at each end of the arc indicates that the link may be followed in either direction.

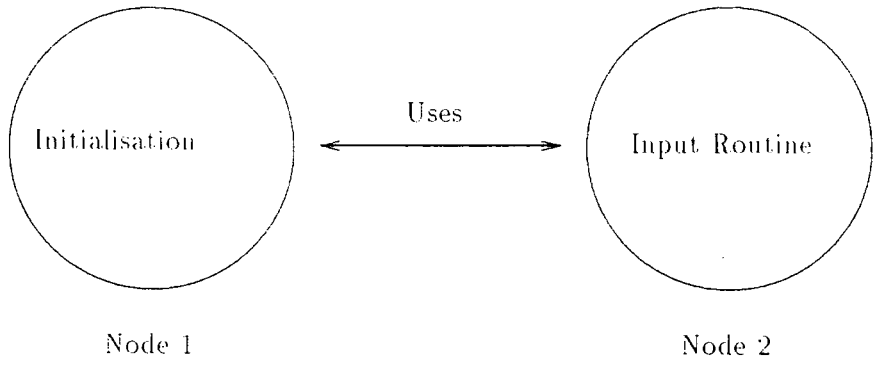


Figure 4.1: Linked nodes in a hypertext

A *graphical* browser can help the user to avoid disorientation, a common problem with large hypertexts, by permitting the traversal of the links between the nodes, the current position in the network being highlighted, using, e.g. inverse video. The author conceives a graphical browser which can operate at two levels, the upper level showing the hierarchy of modules in the software system, and the lower level showing the network of nodes containing information concerning a particular module.

The information in the nodes of a hypertext can be accessed by the usual way by the use of a mouse, clicking on the node of interest. A traversal history can be maintained, i.e. a trail of the links through the hyperdocument, so that, e.g. an audit trail may be implemented. Hypertext provides an easy means of tracing references; machine-support for all link tracing means that all references are equally easy to follow, either forward to their referent or backwards to their reference. In addition new references can easily be created. Networks can be built by users either by starting from scratch, or an existing network can be enhanced by annotating with comments, without changing the referenced document. The user is supported in having several paths of inquiry active and displayed on the screen at

the same time, such that any given path can be retraced to the original task.

4.2.4.1 KMS - Knowledge Management System

Objectives

KMS [1] is a large scale hypermedia system for collaborative work. It aims to help organisations manage knowledge by reducing the effort required to build and maintain corporate databases, since these activities are often the principal bottlenecks in many uses of computers.

Architecture

KMS relies on a wide-area networks of workstations. No other information is given concerning the architecture.

Functionality

The KMS database consists of a set of interlinked screen-sized workspaces called frames - these contain any combination of text and graphics, each of which may be linked to another frame, or used to invoke a program. Conventions exist for the format of frames: the *layout* of the frame *is* the data model.

Features of KMS

1. Navigation

   Achieved using mouse - a linked frame is displayed in the same window.

2. Editing frames

   This can be done at any time.

3. Invoking programs

   Large conventional programs can be run from the operating system shell.

4. Context-sensitive cursor

   Operations available depend on whether the cursor is within the text or in free space.

5. Unified command set

   Cut and paste operations (a set of related commands) are unified into a move command - text can be picked up and repositioned within a frame or can be dragged from one window to another.

4.2.4.2 Dynamic Design

Overview

DynamicDesign [11] is a CASE environment, based on hypertext. Information structures are used for storing source code and the environment has utilities for manipulating the information; the role of hypertext is as the data model.

Objectives

It aims to help in manipulating source code.

Architecture

It has a layered system architecture which allows for extreme modularity and independence of software components.

Functionality

The environment possesses the following features.

1. It uses the Hypertext Abstract Machine (HAM) in a layered system architecture
2. It stores source code, requirements, documentation in a hypertext database, using information structures
3. It allows arbitrary structuring of information and keeps a complete version history of information and structure
4. Nodes in hypertext database contain project components
5. Links relate nodes
6. Nodes and links have attributes
7. Node attribute is Project-Component (Identifies *type* of Project Component contained in node)
8. Link attribute is Relation (Shows the *type* of relation the link provides)
9. Utilities in Dynamic design deal with information structures in the source code context, for example a source code tree

(a) Source browser

The browser is the part of the environment that helps in understanding and maintaining the source code and its auxiliary documentation

(b) Graphbuild

A hypertext source graph is assembled using the program's call tree.

4.3 Discussion

4.3.1 The role of the process model

Riddle [110] has shown the value of a process model is that it is concerned with rigorously defining, analysing, and predicting the impact of software processes with respect to organisational or project-related needs and takes the view that software process models make possible:

1. Effective communication about software processes, involving people and resources
2. Use/reuse of a software process in different situations, since the process model is different for different types of maintenance
3. Maturation and evolution of a software process, by mapping the process to a conceptual schema

4. *Management* of a software process, through the use of people and resources

Stenning [120] views an environment as the effective means of supporting an effective process; the importance of process models is that the quality of the product is determined by the process producing it, and so there is a need to understand and compare software processes and to evaluate and reason about them, so that better ones may be designed and produced, thus improving the quality of the product. The extension of this line of reasoning is that processes should be enactable and therefore should take the form of *programs*, i.e. process programs. Tully [125] describes process programming as:

1. A powerful new form of programming

2. A way of treating existing software systems as programmable resources or virtual machines - in the same way as for example, operating systems or compilers model lower-level programmable resources as virtual machines.

Tully's view is that modelling and programming the software process becomes an experimental test-bed for modelling and programming the human-computer activity in general, for introducing a new and potentially much more highly productive way of programming or system building. Lehman [65] thinks that process programs imply a deterministic development process, which excludes the creative element in producing software. It is the author's view that the development process is probabilistic, but deterministic to a limited extent - it is deterministic in the sense that a software product *will* emerge but probabilistic in the sense that it will almost certainly be deficient in some way, i.e. it is not possible to predict

with *certainty* what the software product can and cannot do.

Lehman doubts that process programs yield more insight into the software development process, or produce better understanding of that process, and offer no significant improvement in the process. According to Lehman, [65] process programming is *one* approach to process modelling - the models so produced being machine interpretable and so can be used as a process control mechanism, e.g. a program-driven mechanism can be used to select and invoke a sequence of IPSE tools. The IPSE could then be tuned to the needs of a particular application of the process, by preparing and loading an appropriate process program or by adjusting parameters in a program which has already been loaded. Lehman asserts [65] that real-time considerations preclude the adoption of this approach.

Notkin's criticism [93] of process programming addresses the argument that since no commercially successful instances of software development environments exist, then it is not possible to construct useful environments through parameterisation, using process programs. Notkin argues that two requirements are necessary before instances can be generated:

1. Experience in building many instances
2. The existence of enough formal notations for the actual parameterisation

Since neither of the two requirements above exist then it follows that instantiations are not possible. There are no instances of *generalised* maintenance support environments so a maintenance process program cannot be used to parameterise an instantiation of such an environment.

4.3.2 Support offered for the generalised maintenance model

Introduction

The only environments which deal explicitly with the phases of the generalised maintenance model, described in chapter 2, are those which explicitly provide support for a maintenance process. The support provided by the environments surveyed for the phases of the generalised maintenance model is discussed below under the respective headings.

4.3.2.1 Verification of need for maintenance

None of the environments surveyed provided any explicit support for this phase of the maintenance model, but those environments which support database queries offer the potential for verifying the need for maintenance, since the maintenance performed on a software system can be stored in a database.

4.3.2.2 Understanding

There are two prerequisites for program understanding: tools to aid in information-capture, and support for abstraction, views, and the creation of information structures. The organisation of information is vital to the process of understanding, since it permits the *organisation* of information into knowledge. For example, source code fragments and their associated documentation need to be juxtaposed, for easier understanding, and a suitable information structure would make this possible.

Tools to aid in information-capture

The analysis of the generic tool classes and functions for information capture and processing is undertaken in chapter seven. Here the author is simply concerned with the presence of a toolset in an environment which could be used to aid information-capture.

Non-hypertext environments

Aspect and Eclipse are open environments, built 'on top' of the host operating system, so they are able to use the tools of the host operating system and can also incorporate third-party tools. Marvel is able to use the tools of the host operating system, but does not aim to incorporate third-party tools. Microscope and the University of Arizona's environment have their own static and dynamic analysis tools. Genesis has as its main tool the information abstractor, which extracts relational information among the software entities of programs. The GSA's environment has a comprehensive range of static and dynamic analysis tools for program analysis.

Hypertext environments

Hypertext environments have addressed the creation and browsing of information structures, but these environments have a very limited toolset - most of the information comes from a cross-referencer: no support is offered for dynamic analysis tools, or for versioning.

4.3.2.3 Support for abstraction, views, and the creation of information structures

This is aided by the incorporation of a query facility, which can establish relationships between objects. The *organisation* of captured information a vital part of the conversion of information into knowledge, and an information structure is a pre-requisite to this organisation.

Non-hypertext environments

Aspect and Eclipse have object management systems, which support abstraction, views and queries. Microscope provides a filtering mechanism to support abstraction, and its knowledge base provides the means of obtaining views of software. The University of Arizona's environment provides no mention of facilities for obtaining different views of software. The GSA's environment provides no support for the creation of information structures, or the filtering of the output from the tools, using anything other than is possessed by the tools themselves; no provision is made to support abstraction or views, and no query mechanism is provided.

Marvel provides a query facility and makes possible the structuring of information through its process model, and its OMS. Marvel's OMS is object-oriented and the *active* object base defines the object classes and the relationship between objects.

Genesis is rule-based and provides the means for defining a software a software development methodology, i.e. a process model, which makes possible the structuring of information.

Hypertext environments

Dynamic Design allows queries can be made based on cross-reference information. KMS

mentions no support for queries.

4.3.3 Modification

Modification of software embraces changes to the source code and the documentation of the system. All the environments surveyed provided support for the modification of the source code, through the use of an editor, and a compiler, and those environments which were not language-specific were open and so could make use of the appropriate syntax-directed editor.

Modification of the documentation, e.g. requirements, specification, and design documents, could be achieved through the use of an environment's OMS. The OMS is necessary to select those documents which need to be changed, and to specify any new relationships which exist between them after the modification has been made to the software. Those environments which have access to an underlying operating system, e.g. Unix, which possesses tools for version and configuration management, e.g. MAKE, RCS, can support this aspect of maintenance.

4.3.4 Revalidation

Only those environments which explicitly support the maintenance process, provide support for this phase of maintenance.

The main activity during the revalidation phase of maintenance is regression testing which makes use of a suite of test cases. An OMS helps to maintain this suite of test cases which was assembled during the development of the software.

4.4 Summary

The role of a software engineering environment in achieving gains in productivity, through the provision of automated support for the maintenance process, has been briefly described. The features possessed by the environments surveyed are summarised in Table 4.1. The literature survey revealed that some software engineering environments provide support for a particular facet of the maintenance process, for example, program understanding or configuration management, but none offer comprehensive support for the *complete* maintenance process. The environments surveyed have no common purpose as regards their aims and objectives, and only the prototype environment of the University of Colorado, Marvel, Genesis, and Eclipse have an underlying process model. In addition, some environments are language-specific and have no underlying process model.

Those software engineering environments which claim to offer support for the complete software life-cycle only offer *partial* support for the maintenance phase of the software life-cycle, that is, they offer support for the *enhancement* of software, which involves an iteration

Summary of Environments Surveyed					
Name	Type	Architecture	Interface	Process Model	Prototyping Support
Microscope	Language-centred	Layered	WIMPS	No	Yes
ASU	Language-centred	Layered	WIMPS	No	Yes
CU	Method-based	?	?	Yes	Yes
Genesis	Language-centred	Layered	WIMPS	No	Yes
GSA	Language-centred	Layered	WIMPS	No	Yes
Marvel	Method-based	?	WIMPS	Yes	Yes
Aspect	Method-based	Layered	WIMPS	Yes	Yes
Eclipse	Method-based	Layered	WIMPS	Yes	Yes
KMS	Method-based	Layered	WIMPS	No	Yes
Dynamic Design	Method-based	Layered	WIMPS	No	Yes
Key					
?	no information				
ASU	Arizona State University				
CU	Colorado University at Boulder				

Table 4.1: Summary of Environments' Features

of the overall software life-cycle. Other environments offer support for the correction of bugs which occur during the *development* of software. An example of such an environment is the IPSE. While it is apparent that an IPSE can offer support for the *managerial* aspects of maintenance, current IPSEs offer little in the way of *technical* support for maintenance: in particular they offers no support for the maintenance programmer, when he is faced with the task of achieving the pre-requisite understanding of poorly-structured and poorly-documented 'alien' code, lacking the high-level design decisions made during the inception of the software project. The 'maintenance' for which IPSEs provide support is not true maintenance, since it is carried out *prior* to release of the software: maintenance is defined [57] as a *post* release activity. IPSEs do not address the problem of extracting information from the documentation of a software system, e.g. the source code, *prior* to its collation and deposition in the database.

The object management system of an IPSE does, however, offer powerful support for the extraction of information from a database concerning the source code of a program under investigation, providing a query facility which can provide different views of software. The facility to manage and manipulate information, providing multiple levels of abstraction and different views of the software, is more important than the toolset possessed by that environment, since 'open' environments can incorporate third-party tools suitable for software maintenance.

Most non-IPSE environments only offer *incomplete* support for a particular facet of the maintenance process, for example, support may be offered for the extraction of information from the source code, but these environments have not solved the problem of the extraction of

the in-line *documentation* linked with the section of source code under investigation. This is important, since alterations to the original source code often mean that the juxtaposition of a segment of code and its associated in-line documentation is not preserved. Attempts to link source code and associated documentation have been made by Garg [42], using a hypertext environment; the primary function of Garg's hypertext environment is the documentation of the software process.

It is clear from this literature survey that there is a need for an integrated software engineering environment to support the maintenance of software, starting from the change request and culminating with the new release of the software.

Chapter 5

The information requirements of a maintenance organisation

5.1 Introduction

The purpose of this chapter is to indicate specifically where the ISMSE can be of help to a hypothetical generalised maintenance organisation, using the maintenance model derived in an earlier chapter. The keeping of a log of the activities of a maintenance organisation *during* a maintenance assignment provides the means for the archival of information concerning the

assignment. The keeping of a log of *all the maintenance assignments carried out on a software system during its lifetime* generates an archive which the author terms a *'maintenance-history'* (analogous to a 'medical history').

5.2 The role of the Maintenance Model

5.2.1 The structure of the Maintenance Organisation.

As pointed out by Foster [40] a model of the maintenance organisation refers to *roles* which are performed by people and as such is a useful abstraction, since it makes it possible to apply the model to teams of disparate sizes; since no assumptions are made about mapping of duties to actual people. The maintenance organisation may consist of a team, in the case of a large maintenance task, or may only be one person, in the case of a small maintenance task; in this latter case this one person must perform the functions of each of the different team members since the maintenance model is *independent* of the size of the maintenance task.

The *structure* of a maintenance organisation is designed to fulfil its role. The strategy adopted for dealing with users' requests for program maintenance is ultimately based on the model of the maintenance process derived in chapter 2; the basis for the strategy is the adoption of a hierarchical structure, shown below in Figure 5.1. In deciding what the

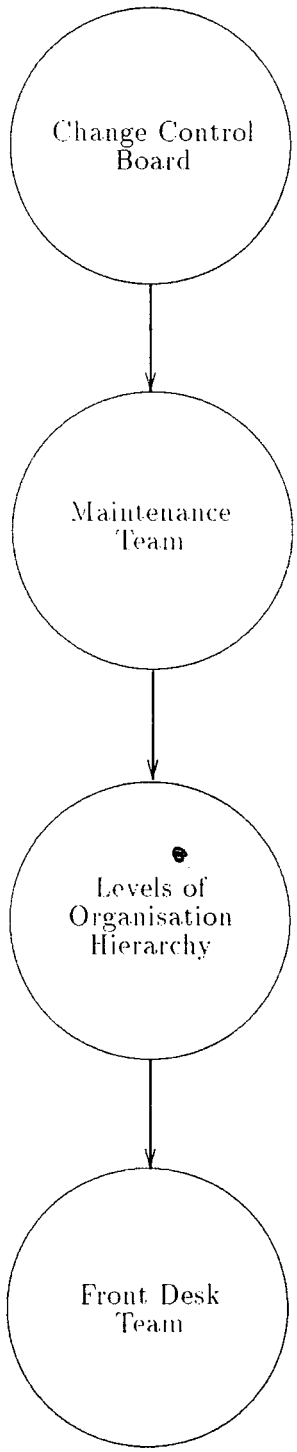


Figure 5.1: Structure of Maintenance Organisation

information requirements of a maintenance organisation are, it is informative to consider the role of the maintenance organisation in determining a mechanism for making a change to operational software. The following is intended to serve as an illustration of the separate contributions to this mechanism by the component parts of such an organisation, when users perceive the need for maintenance on software. The information requirements of the *component* parts of the maintenance organisation differ according to the context in which they function, i.e. the phase of maintenance currently being undertaken. In an earlier chapter a maintenance model was derived which identified four phases:

1. Verifying the need for maintenance
2. Understanding the program
3. Modifying the software, i.e. the program *and* its associated documentation
4. Testing the modified program

Before a strategy can be devised for the storage of captured information, the *type* of information that the environment will process needs to be elucidated. In the following sections the information requirements for each *phase* of maintenance are amplified, to aid in this process.

5.2.2 Verifying the need for maintenance

As mentioned earlier one of the main aims of the ISMSE is to increase the productivity of the maintenance personnel: a maintainer will be *effectively* more productive if he does not have to perform *unnecessary* work on software whose shortcomings are well-known and for which solutions have already been found. Existing environments which provide support for the maintenance process do not attempt to deal with verifying the *need* for maintenance. Maintenance is essentially a user-driven activity, apart from preventive maintenance, which is initiated by the maintenance organisation itself: because of this, requests for maintenance from users must be submitted in a form that facilitates a quick interpretation of the users' wishes. Any ambiguity in this respect will mean time-consuming delays in clarifying with the user exactly what is required, and will adversely affect the productivity of the maintenance organisation. The design of the mechanism by which users may request a change to operational software is a crucial facet of the working of the maintenance organisation.

The request for maintenance should ideally be submitted in a form that can serve as an agenda for discussion by the maintenance organisation, and should be submitted on a form whose design is the responsibility of the maintenance team. This formalisation of the change request will make it easier to devise a strategy for the maintenance assignment including the resources required (men, tools, facilities) and a schedule for its completion. The change request should classify the type of maintenance required and the information appropriate to each type, as described below.

1. Corrective maintenance

Apparent deficiencies apparent in the working of the software which make the modification necessary, including, if possible, a fault report detailing the malfunction of the software, together with hard-copy showing the inputs and the faulty output from the software.

2. Adaptive maintenance

   Details of the type of change in the data or processing environment.

3. Perfective maintenance

   Description of the deficiency in the performance, quality, standards, or maintainability of the software.

4. Enhancement

   When an enhancement is required to software, then the following is required:

   (a) Requirements analysis

   (b) Functional specification

   (c) Task scheduling

   (d) Design analysis

   (e) Design review

   (f) Review of proposed code changes

5.2.2.1 Information requirements for the front-desk

The information requirements are determined by the role of the front-desk personnel, who function as the *interface* between the users and the maintenance organisation.

One of the obvious benefits of a hierarchical type of structure is the ability of one level in the hierarchy to act as a filter for the level above. This is a useful strategy with organisations which have dealings with customers (users), especially when verifying the *need* for maintenance.

The idea of a 'front-desk' in a maintenance organisation which acts as a filter for maintenance requests has been suggested by several authors, e.g. [83, 40]. This 'front-desk' is responsible for the first action taken on receipt of a request for maintenance, and so the information needs of the 'front-desk' personnel differ markedly from that of the rest of the organisation.

As well as having answers to known problems concerning a software system, the 'front-desk' personnel need to have access to a synopsis of the 'maintenance-history' of the software, including all known versions, variants and impending releases. (A version is defined [57] as the latest instantiation of a software system which has superseded all other instantiations, a variant is defined as an instantiation of a software system which temporally co-exists with other instantiations of a software system.)

Users can then be referred by 'front-desk' personnel to solutions for problems, or informed of new releases, that have alleviated or solved such problems. In the situation where a

maintenance organisation is confronted with an old software system, which has suffered degradation, lacking reliable documentation and a 'maintenance-history', the front desk is unable to function in this way since the 'maintenance-history' commences after reverse engineering and re-engineering have been performed on it. The software system then begins a new lease of life with a 'clean sheet'. Those requests that cannot be filtered out are referred to the next level in the hierarchy of the maintenance organisation. User manuals concerning software which is still being supported would enable the front desk team to decide whether the user's perception of a problem is in fact real, or is a result of misinterpreting how the software will behave under a given set of conditions.

Mellor's idea [83] of *users* being able to interrogate a database containing information concerning versions and variants of a software system would *effectively* further enhance the productivity of the maintenance organisation as a whole. Typical of the information required by 'front-desk' personnel are answers to questions such as:

1. Has this bug been corrected on a previous occasion ?
2. Are there problem reports concerning a particular system already on file ?
3. Is there a subsequent version of the software in a new release ?
4. What is currently being maintained ?
5. If the fault can not be easily rectified, e.g. when the specification is correct and matches the code but the requirements are incorrect then an 'avoidance' action for software with a usability problem should be investigated, to prevent the manifestation of the bug.

Requests for maintenance must be assigned a priority by the maintenance organisation, which implies that a queueing system is necessary. If the request concerns a problem which has a trivial effect on the working of the software and is difficult to fix, it is assigned a low priority and vice-versa. Having submitted a request for maintenance the user naturally wishes to monitor the progress of his request, in view of this the information concerning the priority assigned to a maintenance request, including the results of preliminary investigations, needs to be kept available so that front desk personnel can give *meaningful* answers in response to queries from users. The maintenance organisation must be able to change the priority of jobs in the queue and have the means of updating the queue.

5.2.3 Understanding the program

Understanding source code was discussed in chapter two in the context of the maintenance model. This understanding is on two levels: the maintainer must understand what the program is doing as it produces the incorrect output, this may be possible from the fault report as documented in the change request. If this information is not available then the maintainer must attempt to duplicate the conditions which gave rise to the apparent abnormal behaviour. The maintainer must also understand how the program *should* work when producing the correct output.

To maintain software, it is necessary to know what it does, how it does it, and how it

can be safely modified. The backbone of maintenance is program-understanding; much of the information collected is to be used to achieve this primary goal but the *use* of this understanding to modify the software is the secondary goal. Understanding is the key to knowledge, and the acquisition of knowledge is aided by:

1. A storage schema that allows fast retrieval of the captured information.
2. Organisation and management of that information, so that it may be processed into knowledge, the processing being achieved by human intervention.

Although program-understanding can be achieved from a study of the source code alone, in the first instance, the environment must support the input of *supplementary* information from the maintainer which may have been gained manually or deduced from other information.

In the context of the time taken to complete a maintenance assignment, program-understanding is the rate-determining step, since the understanding of a software system occupies most of the time spent on a maintenance assignment [104]. The time taken to gain this understanding is directly related to the quality of the documentation, being aided by good quality documentation and vice-versa.

The strategy for developing an understanding of alien code rests with the maintenance team and cannot be imposed by the environment, instead the environment must offer support for the execution of this strategy. The ultimate goal of program understanding is three-fold:

1. To establish a high-level understanding of the program, from its structure

2. To discover how each module of the program plays its part in achieving the objectives of the program

3. To obtain a detailed understanding of the program, within a module, at the statement level, i.e. a 'local' understanding of the code which is of immediate concern to the proposed modification.

A query facility offers great utility since it offers the means for the maintainer to build information structures thus converting *data* into information: information is something that enables us to increase our knowledge: this is an iterative cycle. The kind of tool which can act as a query facility is dependent upon the conceptual schema chosen for the data structure which is deposited in the database. The acquisition of information for the purpose of understanding software follows a cycle analogous to the software life-cycle. This provides the opportunity for refinement of the model of the understanding of the behaviour of the program: iteration of this cycle with generation of successive hypotheses and their subsequent testing offers the surest route to understanding. The stages in this information-acquisition cycle are shown below:

1. What needs to be understood ? (Requirements)

2. What information is needed ? (Specifications)

3. Design an information structure to hold the information (Design)

4. Issue high-level instructions to obtain the information (Coding)

5. Does the information structure aid in understanding ? (Testing)

At present the task of gathering this information is largely a manual one, and unless the core of the environment, i.e. the database and the toolset, can be fully-integrated, the environment will not be able to function as effectively as it otherwise could. Integration will not be addressed in detail in this thesis. There are software tools available from vendors which can provide the maintainer with valuable information about a piece of software, these are discussed in the next chapter.

The maintainer must be able to *manage* and interpret this information in such a way as to improve his understanding of the program: this understanding of the software is determined by the *quality* of the information available to the maintainer. Developing an understanding of software is an iterative process: in the domain of program-understanding a hypothesis is generated and this must be tested: when the boolean condition (predicted behaviour = actual behaviour) evaluates as true, the cycle is terminated. When testing a hypothesis, for a given input to the program the maintenance programmer must be able to predict:

1. modules, routines invoked
2. changes in values held in variables
3. the output of the program

These predictions must be confirmed either through the use of an execution-flow-trace tool and a debugger, or by running the program to verify that it behaves as predicted. If the cycle is not terminated, the hypothesis must be corrected and the process described immediately above must be repeated. The environment must support this method of working, i.e. the

environment database must be able to store successive versions of understanding leading to a 'complete' understanding of the software. The environment must enable the maintainer to find answers to his questions and must facilitate the rapid testing of hypotheses since 'feedback' is the key to learning, (i.e. increasing understanding), and 'feedback-inhibition' hinders the learning process. Typical information gathered and stored to facilitate program understanding includes:

1. High-level documentation containing relevant information pertaining to the *development* of the software which requires maintenance, i.e. requirements, specifications, high-level design-decisions and, if available, the underlying philosophical goals. Often high-level information can be very important, for the following reasons.

   (a) The original requirements document would help to decide whether the users' expectations of the software are realistic, i.e. is there a need for maintenance ?

   (b) The time required to achieve the necessary understanding of the software can be considerably shortened.

   (c) High-level information concerning the original development of the software can be very useful when attempting to gain an understanding of the program at the statement level.

   (d) High-level functional specifications can help to decide whether the code is 'correct'.

2. A call-graph showing the current program structure with respect to the calling relation between routines, and the intended structure of the amended program.

3. The source code itself, both the structured and unstructured versions, if a re-structuring

tool has operated on 'monolithic' code.

4. The results of analysis of a database object (cross-reference listings, etc.), or objects produced by transformation tools, e.g. the recovery of the design from a re-engineering tool, as well as a high-level description of what each module does.

5. In-line documentation regarding the code, containing information concerning the design of the software structure. If this is in a separate file then an editor can help with finding information - however, 'in-line' comments embedded in the code may not be so easy to find - there is a need for a tool that can help with this.

6. There are four main kinds of lower-level documentation.

   (a) User documentation - how to use the program

   (b) Operations documentation - used to direct the execution of the program

   (c) Program documentation - how the program works

   (d) Data documentation - data model and data dictionary

   Since variables are used in booleans, which are used to determine the flow of control within the program, it must be possible to specify that links should be made between information about data flow and control flow.

7. Tools available to the maintenance programmer and their functions; this should include a precise summary containing:

   (a) which version of the tool is being used

   (b) how the tool is invoked

(c) description of a tool's function - including switches which can function as filters, so that abstractions can be obtained

(d) limitations of the tool and any known 'bugs'

(e) which tools can be used together and how they can be used together (as in Unix)

The provision of the means of *organising* information gained about a system is vital because, for example, of the existence of delocalised plans in the code [67]. Storing the related parts of a plan in the environment's database, enables the maintainer to build the most suitable information structure to facilitate this understanding. In essence the information is processed into knowledge by making it part of an information structure, and is required regardless of the type of maintenance being carried out.

5.2.4 Modification of Software

The maintenance of software, *including* documentation, is often carried out under great time constraints, and often there is little time left after a modification has been made to source code, because of pressure on the maintenance organisation to tackle other assignments. This often means that there is a failure to redocument the software fully and accurately, as a result of this there is a 'knock-on' effect, which has profound implications for the future maintenance of the software: the mechanism for this is now briefly described.

If redocumentation has been done badly then when the software has to be maintained again

the documentation will be seen as unreliable and the whole process of gaining an understanding of the software system *from the source code* has to be repeated. The hindering of program-understanding in this way causes the maintenance team to be placed under time pressure, which again means that it is more likely that redocumentation will not be done as well as it should, since it is usually done *after* the modifications to the source code. The cycle is completed and is destined to be repeated, establishing a 'vicious circle'; to break out of this cycle requires that redocumentation is not a neglected activity, but assumes paramount importance before and during the modification activity, as it does during maintenance of, e.g. safety critical systems. This desirable state of affairs is more likely to come to pass if the productivity of the maintenance team can be improved, since more time will then be available to carry out redocumentation. One of the aims of the work in this chapter is to suggest tools and methods which will help to achieve this improvement.

It should be pointed out that good quality maintenance of documentation is more likely to be achieved by making it easy to do, rather than by the use of coercion, or merely by providing time for this activity. Good quality maintenance of documentation is likely to safeguard the gain in productivity, achieved by using software tools, and will assist the efficiency of future maintenance efforts. Poor quality redocumentation will mean that the use of tools and methods to increase productivity will have been largely wasted. As part of the documentation reflecting the changes made to the program the original source code and the modified source code should be placed 'side by side' with a mapping to show the changes in the program hierarchy.

The formalisation of the approach to the documentation of the maintenance process, in

conjunction with the use of tools and methods provides a framework, within which the maintenance activities, and the information associated with them are recorded; the type of recording is discussed in the next section. Documentation of maintenance is more likely to be of good quality by adopting a rigorous approach, and by carrying it out *as maintenance is being performed*, so that it is a by-product of the maintenance activity. Information requirements for program modification include:

1. The language and version in which the source code is written, is needed to ensure a clean compile when changes are made to the source code.

2. The source code.

3. The new requirements and associated functional specifications together with the program design.

4. The place in the program where the change(s) are to be made must be decided, using, e.g. the techniques of inspection and walkthrough.

5. The results of impact analysis must show that any changes made to the software are free from adverse side-effects. To aid in this respect a library of potential causes of side-effects can be stored in the database [41].

5.2.5 Revalidation

For all types of maintenance, before the modification can be implemented by the maintenance programmer, the change approval board must have indicated its approval of the proposed change(s) and information concerned with quality assurance must be to hand, such as continuity of programming style, documentation update and audit trails.

The technique used to ensure that no adverse changes have been made to software during maintenance is known as regression testing.

5.2.5.1 Regression Testing

IEEE [57] define regression testing as:

'Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements.'

Included under the heading of regression testing are:

1. System testing

   The tests and their descriptions together with the corresponding expected results should be entered into the ISMSE database. The tests should be carried out in conjunction with the users of the program, and their comments concerning the changes should be archived in the ISMSE database, along with information concerning the

change(s) to the user manual. This is part of the redocumentation process.

2. Unit testing

   Each modified unit should first be tested in isolation.

3. Integration testing

   Any enhancements that have been made to the software should be incorporated using the appropriate integration strategy, e.g. 'top-down' or 'bottom-up' and the results associated with each phase of the strategy stored in the ISMSE database.

Test cases, either those included as part of the documentation when the software was developed or those produced by the maintenance team, (where alien code lacks documentation), must be run to establish as far as possible, the absence of any undesirable side-effects, using appropriate testing strategies.

5.2.5.2 Testing Strategies

There are two types of testing strategies:

1. Structural testing

   This uses knowledge of the program's construction, e.g. branch testing ensures that a test-data set executes the outcomes of all branches in the system. If the source code is available then this strategy can always be used.

2. Functional testing

   This uses the system's requirements to derive the test data. If the requirements documentation associated with the source code is deemed not to be reliable then this strategy cannot be used.

Corrective maintenance requires that an error condition be reproduced to confirm the existence of a deficiency in the software. This is achieved by devising an appropriate test-case, which must then be added to the test-suite to ensure the absence of the error, after the modification to the source code. If the maintenance on the system requires that a change is made to the requirements, i.e. perfective, adaptive, preventive maintenance, then new functional tests will be needed. Corrective modifications do not change the system's requirements and so no new functional tests are required. The two major problems in regression testing concern test-suite maintenance and revalidation strategies.

5.2.5.3 Test-suite maintenance.

Corrective or perfective maintenance on source code will probably cause structural changes, necessitating new structural tests to be devised. In addition, existing tests may become redundant and may need to be eliminated from the test suite. Identification of these redundant test-cases is problematical. Regression testing uses a test-suite, which includes test input data and the resulting test output: current regression techniques use a functional strategy, i.e. every system modification is accompanied by a re-run of the test-suite, with a comparison of the corresponding outputs.

In a file-based system, the results of executing the test cases, or test scripts are represented as output files. Any output file whose content differs from the expected output is flagged to see if it represents an error condition. The maintenance test-suite should be based on that devised during the development process. In addition, those new tests devised during maintenance should be added to the test-suite, reflecting the updated system specification and those test-cases which have become redundant then eliminated. To achieve this means that the entire test-suite must be executed: for structural tests this involves constructing a table that associates test-cases with a program's structural elements. Those test-cases which execute identical structures are reduced to one instance; a 'spin-off' benefit associated with this approach is that structures which are not executed by any test-cases are revealed.

An alternative approach [68] is the selective analysis of those changes made since the last analysis; only those structural tests which execute the changed structures, or the routines containing those structures are re-executed and analysed: this indicates those tests which were associated with those sections of code since modified, and which are now obsolete. An automatic approach to the identification of obsolete functional tests is only possible where a formal specification language has been used, which permits the analysis of the changes in the specification. The use of a function table has been proposed [68] listing all the functions with the numbers of the functional test cases that execute them.

5.2.5.4 Revalidation strategies

Executing an entire test-suite after a modification involves many people, is inefficient and consumes large amounts of time and computational resources. On the other hand testing a system by selecting test-cases intuitively or randomly is unreliable. A strategy for the revalidation of software is required to systematically select those test-cases to re-run after a modification, using control flow and dataflow information as a basis for the selection process, which needs to be stored in the database of the ISMSE.

5.3 Summary

The work in this chapter has presented a coherent strategy for the management of information, in support of the maintenance process. The establishment of a link between productivity, program understanding and documentation has pointed the way towards achieving the main goal of a maintenance organisation, that of an increase in productivity. Keeping a log of the maintenance process means that redocumentation is not done on an 'ad hoc' basis, but is done concurrently with the changes to the code. The documentation is therefore more likely to accurately reflect the changes made to the code, thereby making the future understanding of the software easier, safeguarding the gains in productivity achieved by the partial automation of the maintenance process.

Chapter 6

A high-level design for an ISMSE

6.1 Introduction

The subject of this chapter is the production of a high-level design for the ISMSE, i.e. a design within whose framework a documentation paradigm, the subject of this thesis, can be rigorously specified (in chapter eight), with a view to prototyping this part of the ISMSE (in chapter nine), to demonstrate the feasibility and utility of this paradigm, as a means of increasing the productivity of a maintenance organisation.

6.2 Choosing a Software Development Model for the ISMSE

In terms of the traditional software life-cycle model, shown below in Figure 6.1, the design of a software system relies on a requirements specification of that system. Typically, the

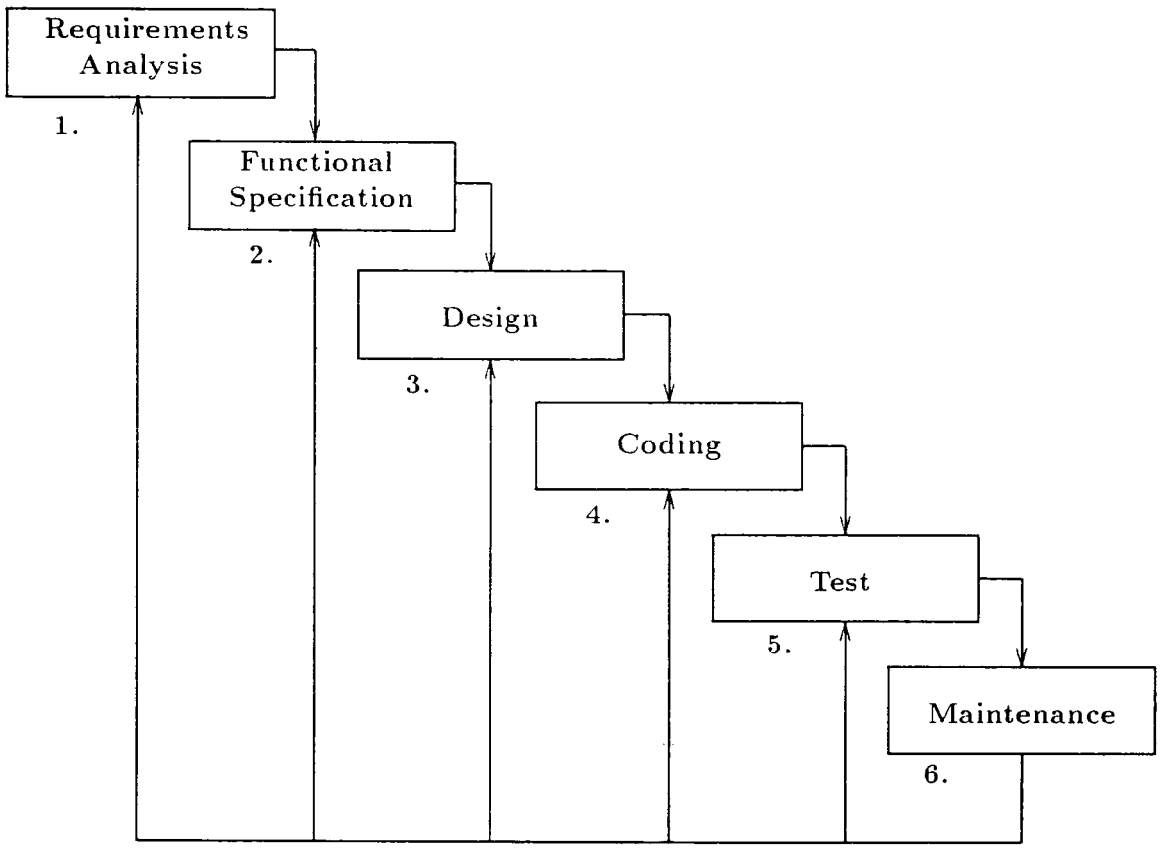


Figure 6.1: The Waterfall Model of the Software Development Life Cycle

time span in building a complex software system such as the ISMSE may be of the order of several years. During that time the requirements may change, and the development life-cycle model must be able to take account of this, so must therefore incorporate some prototyping capability, in order that the users of the system can verify the software system's continued

adherence to their requirements.

The traditional waterfall life-cycle model due to Royce [113] shown in Figure 6.1 is a development of the stagewise model due to Benington [10], which advocated the development of software as a sequence of successive stages, lacking iteration. Royce's waterfall model introduced feedback loops between successive stages, to preclude the expensive revisions needed if feedback were to occur across several stages, and made prototyping part of the software life-cycle, operating in tandem with requirements analysis and design.

There followed many other variations of the waterfall model, enumerated by Boehm [18], and elaborated here in order of their chronology. A variation of Royce's model due to Mills [85], adopted a top-down structured approach and produced a variant of the waterfall model which embraced the concept of 'risk-management'; each stage of the waterfall model including a validation and verification activity to cover high-risk elements, re-use considerations and prototyping. The waterfall model does not support versioning and making software amenable to change: Parnas' concept of encapsulation, through modularisation and information hiding [101] did much to rectify this shortcoming, though this approach does not explicitly support prototyping and reuse. Distaso [35] incorporated incremental development into the waterfall model. Balser [7] provided a conceptual means of incorporating automatic programming, program transformation and knowledge-based software assistant capabilities into the waterfall model, but this model does not offer explicit support for versioning. The advent of fourth generation languages and rapid prototyping capabilities gave rise to evolutionary versions of the waterfall model, e.g. that of McCracken [80], and mixed versions, e.g. that of Giddings [44]. Lehman made use of abstraction, leading to a formal specification, followed by a set of

formal deductive reification steps [64] proceeding through design and into code, rather than using a uniform progression, implicit in the waterfall model. The approach has not been evaluated with respect to versioning or reuse. The models which are generally accepted as being most acceptable for the development of large software systems are the spiral model due to Boehm [18], and the Software Process Maturity Model (SPMM) developed at the Software Engineering Institute at Carnegie Mellon University in Pittsburgh, U.S.A. These models are discussed below.

6.2.1 The Spiral model.

The spiral model of software development and enhancement shown in Figure 6.2 due to Boehm [18] favours a risk-driven approach to the software process, rather than a strictly specification-driven or prototype-driven process, being based on the strengths of other models, while at the same time minimising their shortcomings: Boehm's spiral model includes most of the previous models discussed as special cases, and provides guidance as to which of these previous models best fits a given software situation.

Referring to Figure 6.2 the radial dimension represents the cumulative cost in accomplishing the steps to date. The angular dimension represents the progress made in completing each successive cycle of the spiral.

The model holds that each cycle of the spiral involves a progression through the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from

an overall concept of operation document down to the coding of each individual program.

6.2.1.1 A typical cycle of the spiral

Each cycle of the spiral begins with the identification of the objectives of the portion of the product being elaborated (performance, functionality, ability to accommodate change), and the alternative means of implementing this portion of the product (design A, design B, re-use), together with the constraints imposed on the application of the alternative (cost, schedule, interface).

An important feature of the spiral model is that each cycle is completed by a review involving the primary people or organisations concerned with the product. This review covers all of the products developed during the previous cycle, including the plans for the next cycle, and the resources required to carry them out. The major objective of the review is to ensure that all concerned parties are mutually committed to the approach to be taken for the next phase. Only in this way can the risk, and ultimately the cost, of the project, be minimised.

The plans for succeeding phases may also include a partition of the product into increments for successive development, or components to be developed by individual organisations or persons. Thus, the review and commitment step may range from an individual walkthrough of the design of a single programmer component, to a major requirements review involving developer, customer, user, and maintenance organisations.

Boehm's spiral model advocates keeping the spiral as tight as possible, so that if a mistake

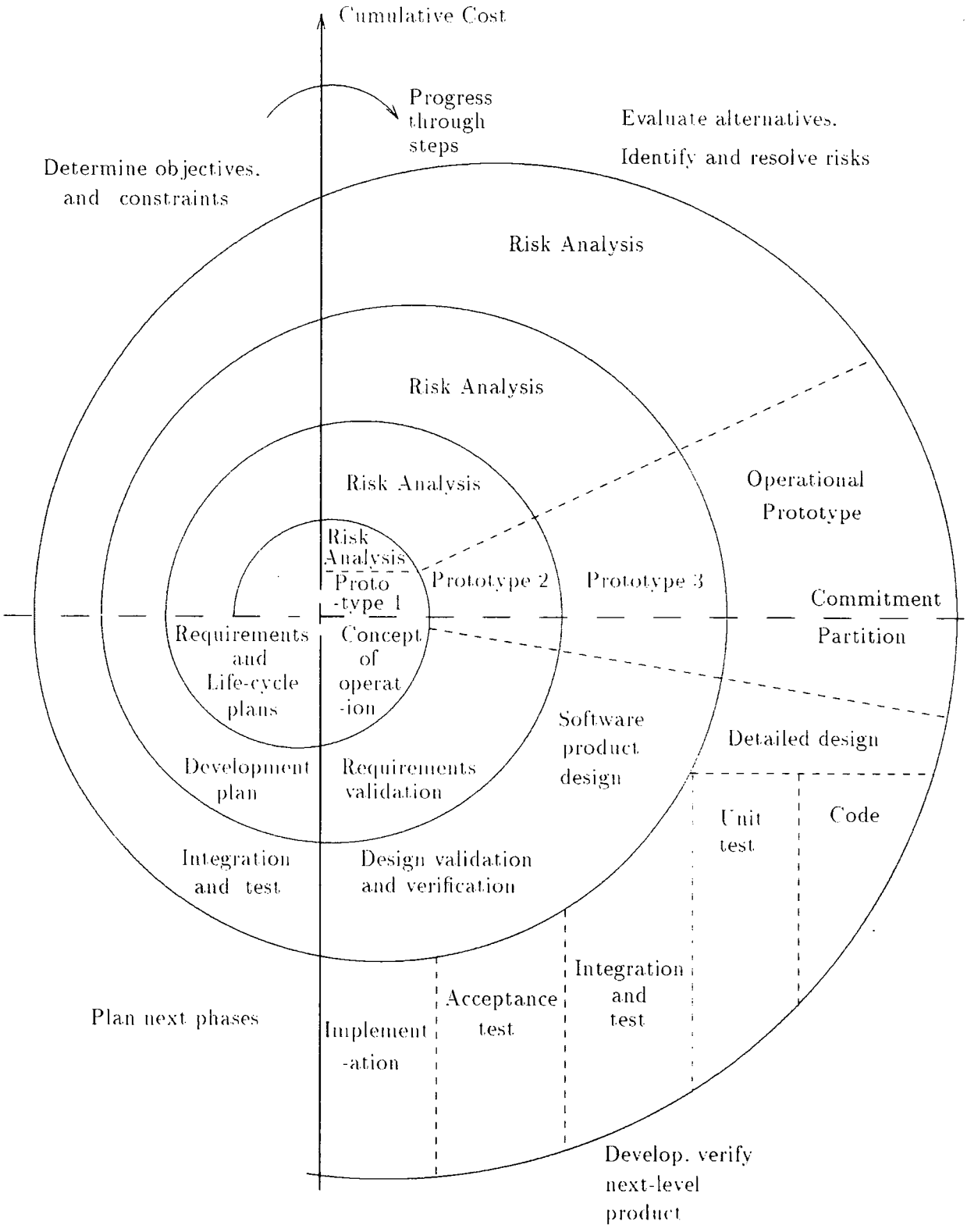


Figure 6.2: The Spiral Model of the Software Development Life Cycle

does become apparent, during the review at the end of the spiral, then not as much back-tracking needs to be done, and so the expense is minimised. The spiral model applies equally well to development or enhancement effort.

Commonality between the Spiral Model and SPMM

The SPMM framework shares with the Spiral model the goal of aiding the management of the software process, embracing the concept of 'risk', its analysis, and a mechanism for incorporating software quality objectives into software product development. The main reason for this risk-driven approach is that mistakes become much more expensive to rectify at lower levels, e.g. during the coding phase, than at higher levels, e.g. during the requirements phase. Unlike the Spiral model the SPMM provides a 'yardstick' for an organisation to assess the degree of precision with which it manages its software process.

6.2.2 Software Process Maturity Model (SPMM)

Assessments of the capability of a process in producing a high-quality software product are often based on the Software Process Maturity Model (SPMM), developed at the Software Engineering Institute of Carnegie Mellon University in Pittsburgh. The model is shown in Figure 6.3 below. According to Arthur [6] this model is gaining widespread acceptance by the Software Engineering Community, and is under consideration as an ISO standard. The model aims to guide organisations responsible for the production of software, through

increased control of the process used for developing and maintaining software, and through the controlled evolution of the software engineering environment, by cultivating software engineering excellence.

The main benefit of using the SPMM is in the narrowing of the scope of improvement activities; the SPMM identifies five levels of maturity which make possible continuous process improvement. Each level has 'characteristics' and 'challenges'. 'Characteristics' describe the current nature of the process which is responsible for the product, and 'challenges' describe the necessary improvement to take the process up to the next level. The main weaknesses of the SPMM are that:

1. No indication is offered as to how to progress from one maturity level in the model to the one above.

2. No account is made of the needs and goals peculiar to an organisation, which may not map onto SPMM.

3. No guidance is offered as to how to assess the quality of any resulting products from a process, or the suitability of any process model chosen to execute a particular task.

The productivity of the organisation and the quality of the software increase with increasing level, while the risk of producing defective software decreases. Most organisations can only aspire to level 3 in the SPMM, since the financial commitment needed to fund the model further than this is prohibitive; any organisation considering embarking on the production of an ISMSE would certainly need to be at this level.

Productivity and Quality increase

Level 5	Characteristics	Challenges
Optimising	Improvement feeding back into process.	Optimisation at the organisational level.

Level 4	Characteristics	Challenges
Managed	Quantitative. Measured process. Data gathering and analysis.	Changing technology. Problem analysis. Problem prevention.

Level 3	Characteristics	Challenges
Defined	Qualitative. Process defined and institutionalised. Control over software process.	Process management. Process analysis. Quantitative quality plans.

Level 2	Characteristics	Challenges
Repeatable	Intuitive. Processes depending on individuals. Using prior experience in doing similar work.	Training. Technical practice. Process focus.

Level 1	Characteristics	Challenges
Initial	Ad hoc.	Project management. Project planning. CM and QA.

Risks decrease

Figure 6.3: The Software Process Maturity Model (SPMM)

6.3 The design process

Introduction

There are two distinct types of design activity, those of external design and internal design.

1. External design

   The relationship between requirements and design is not straightforward since the boundary between requirements analysis and external design is not well-defined. During requirements definition a design abstraction permits separation of the conceptual aspects of the system from the implementation details, and determines:

   (a) functional characteristics

   (b) data streams

   (c) data stores

2. Internal design

   This concerns the relationships between software components, e.g. modules. Eventually procedures and algorithmic detail are determined.

6.3.1 The role of abstraction

Abstraction mechanisms, i.e. functional abstraction, data abstraction and control abstraction, control the amount of complexity that must be dealt with at any particular point in the design process, by systematically proceeding from the abstract to the concrete, as shown in Figure 6.4 below. The high-level design *process* involves describing the system at a number of different levels of abstraction, proceeding through a number of stages and is an *iterative* process. This design paradigm is used as a basis for producing a high-level design for the ISMSE: the first stage in this process is obtaining the high-level requirements, as a basis for the design. The terminology used to describe this aspect of software engineering is not standardised, the author has adopted the terminology used by IEEE, i.e. the high-level requirements specification as a basis for the high-level design in this chapter is termed the Outline Software Requirements Specification (OSRS). As pointed out by Sommerville, [118] there is a distinction between needs and requirements; needs are very high-level, but requirements are much more detailed. Starting from the premise that there is a need, information about a problem is collected and analysed, leading to a comprehensive problem specification, from which a software solution is designed and implemented. Sommerville [118] has shown that the requirements for a software system exist at different levels of abstraction:

1. Requirements definition

   This corresponds to an *Outline* Software Requirements Specification (OSRS), which will be used to describe the software requirements at a *very high level*; a software requirement is a property that a software system must satisfy. The purpose of an

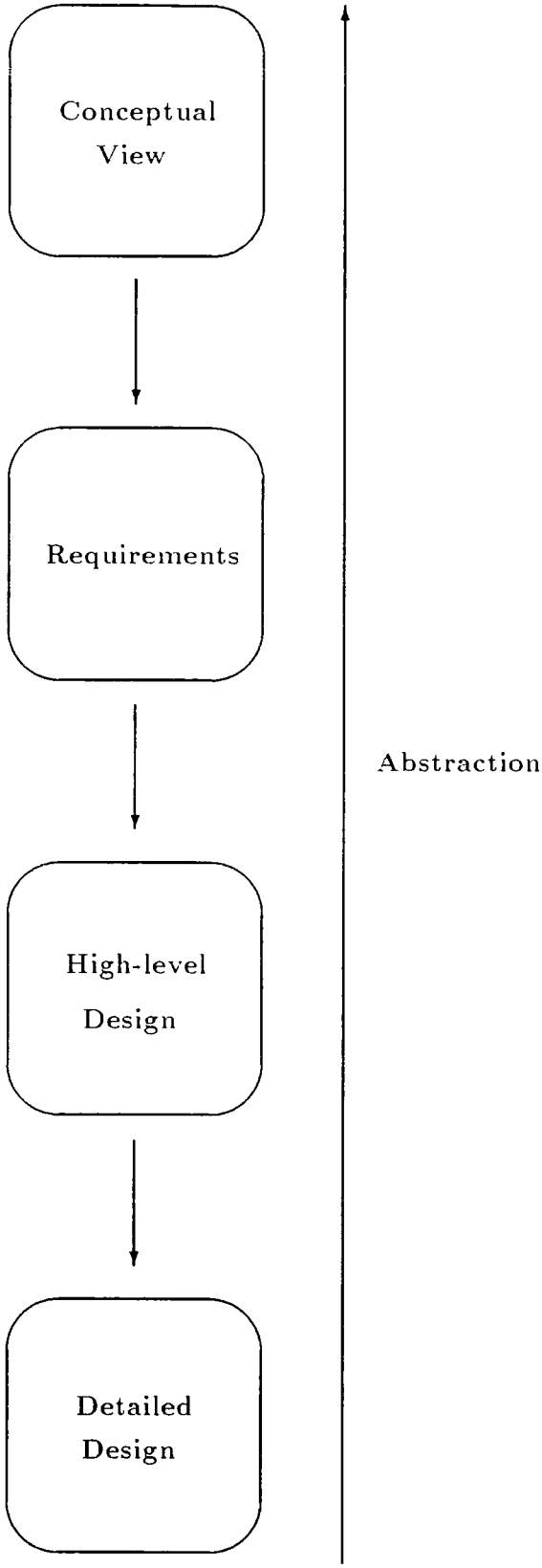


Figure 6.4: Abstraction of the Design Process

OSRS [58] is to define and document a software system with respect to:

(a) functionality, i.e. what the software is to do, not how it is to do it

(b) performance, in terms of e.g. its response time

(c) the design constraints imposed on the implementation, e.g. implementation language

(d) attributes, e.g. portability, maintainability

(e) external interfaces, e.g. interactions with people, and other software, e.g. third-party tools

The requirements definition describes the services to be provided for the user by the system. These high-level requirements are refined using, e.g. prototyping, to give the requirements specification and software specification described below.

2. Software Requirements Specification

This is a *precise* description of the requirements for a software system, defined by Yeh and Zave [131] as "A set of precisely stated properties or constraints which a software system must satisfy". An SRS is sometimes referred to as a *functional specification* and allows a design to be validated using an explicit, formally-specified system model as an aid to understanding the system. Here the notation is more formal since this document may function as a contract between client and user. The services provided by the system are described in more detail.

3. Software Specification

Otherwise known as a design specification, this is an abstract description of the software design, and is intended to serve as a basis for the design and implementation of the

software. The use of formal specification techniques is appropriate since this document is for software designers, not system users or managers.

A structure for an OSRS has been suggested by IEEE [58] and is regarded as an industry standard. This document has a hierarchical structure, and is shown in Figure 6.5 below. It has been pointed out by Zahniser [132] that the IEEE standard for an OSRS requires that Inputs, Processing and Outputs are defined prematurely, since they are concerned with *data*, which is not considered as being part of requirements analysis. Considering these aspects at the requirements definition phase means that a complete data analysis and data-oriented design is undertaken *before* the requirements specification is completed. Moreover, the IEEE OSRS implicitly assumes that the software development method will be the conventional one, using the 'waterfall' model of the software life-cycle, i.e. it will be 'top-down' in nature. Since the aim in this chapter is to produce a high-level design for an ISMSE, a much-simplified version of the IEEE OSRS will be used.

6.3.2 An Outline Software Requirements Specification (OSRS)

This chapter is concerned with a high-level design, which requires only an Outline Software Requirements Specification, enabling a prototype to be built, which can be used to validate and refine the OSRS.

IEEE Prototype OSRS

Table of Contents

Figure 6.5: Structure for an OSRS

6.3.2.1 Obtaining the OSRS

The technique used for deducing the OSRS for an ISMSE is from the point of view of the data, since, as will be demonstrated later in this chapter, the ISMSE is an information sub-system and data/information is its raw material. There exist several methods [58] of capturing and expressing the requirements for a software system:

1. input/output specifications
2. using a set of representative examples
3. the specification of models
4. natural language
5. data flow diagrams
6. structured text
7. data modelling

6.3.2.1.1 Expressing the OSRS

The most common method of specifying an OSRS is to use numbered paragraphs of prose text, i.e. expressed in natural language, within numbered sections, with each paragraph specifying or qualifying some requirement. This should be capable of being understood by non-specialist staff, including the potential users of the system.

There are good reasons for *not* using *unstructured* natural language as a means of expressing the requirements for a software system. The inherent ambiguity of unstructured natural language and the complexity of prose description means that it is unsuitable for expressing requirements clearly and unambiguously. Moreover it is difficult, if not impossible to verify if requirements are complete and non-conflicting, because the length and complexity of prose specifications make them difficult to understand. However, there are two overpowering reasons for using natural language for an OSRS. The first reason is concerned with the *high-level nature* of the concepts which are embodied in an OSRS : no 'closed' formal language, such as a programming language can yet adequately express such high-level software requirements.

The second reason is due to the *imprecise* nature of the requirements for a complex software system which are likely to evolve, since:

1. In the early stages the understanding of a complex system is likely to be flawed, and at a later stage the deficiencies of the requirements become apparent.

2. The high-level requirements for the ISMSE are based on the maintenance model derived in chapter two. Here it was pointed out that the maintenance model adopted was a generalised one, not only because a detailed model of the maintenance process is organisation-specific, but also because the phases of maintenance that involve creativity and understanding are not well-understood. This means, implicitly, that the model of the maintenance process is an evolving entity. This does not preclude basing the ISMSE on a generalised maintenance model, since prototyping has an important role to play in building the ISMSE, discussed later in the section on design.

3. Users will assemble and combine tools in ways which are not as yet predictable.

This highlights the role that prototyping plays in the design process, discussed later in the chapter. When it is not possible to formulate system requirements *precisely* because a system under development is completely new, and not amenable to detailed analysis, then an imprecise statement of the requirements must be formulated and subsequently refined as their shortcomings become apparent.

Natural language is the only realistic means of specifying imprecise requirements, formal notations can *not* be used to describe something which is imprecise. The specification should be structured hierarchically, with different levels of detail, with each entity described in progressively more detail, presenting a clear, complete concise statement of the requirements. Those notations which have been developed to specify requirements are all based on natural language, imposing some structure on the specification, and limiting the natural language expressions which may be used; they also enhance the natural language specification, by means of graphics. Languages which have been used to express software requirements include PSL/PSA [122], SADT [112], and RSL [3]. Notations used to specify external design characteristics include data flow diagrams, structure charts and HIPO (Hierarchy Input Process Output) diagrams. Data flow diagrams are a powerful tool for requirements analysis, and for representing external and top-level internal design specifications, since they can be used at any level of abstraction. The requirements in this chapter will therefore be expressed using a combination of natural language and data flow diagrams.

An OSRS should specify any changes anticipated when the requirements are originally for-

mulated, and give reasons for high-level design decisions made in the light of knowledge of the problem domain, providing an invaluable aid to future maintainers of the software; in addition the document should be easy to change.

6.3.2.1.2 The conceptual model

The highest-level view of a system's *functionality* concerns the *goal* of the system, e.g. improving the maintenance process. A goal can not easily be validated, and so a refinement of the goal into *objectives* is necessary, since there are *requirements* which *accompany* the objectives. The highest level in the abstraction of the design process, corresponding to the goal of the system is the *conceptual model* of the software. The conceptual model is produced from a critical examination of the system's objectives, and the *role* the system performs in achieving these objectives. This conceptual model determines the type of system to be built, and is comprised of two main parts:

1. System Model

   This establishes the entities of the problem domain, their functional characteristics, and the way they are combined to produce an overall system structure.

2. Process Model

   This defines the requirements of the system, i.e. the ISMSE, in terms of the operations performed by it, covered in chapter 2.

The model of the design process places design chronologically after the requirements specification, but, in practice it is not possible [37] to perform a requirements definition without

doing some preliminary external design, which begins during the analysis phase and continues into the design phase: at first the conceptual model is somewhat vague, (the conceptual model for the ISMSE is discussed later in section 6.2.2.1.5). Initially, external design involves refining the OSRS using prototyping as a requirements validation tool, establishing a high-level structural view of the system, with a description of all externally observable characteristics, e.g. user displays, external data sources and sinks. The functional characteristics, and high-level process structure, i.e. the maintenance model, are also determined at this stage, showing the overlap of requirements and design phases. Using the conceptual model, the OSRS for the ISMSE can be derived by considering the *outputs* of each of the major functions of the ISMSE, i.e. from a study of the external behaviour of the ISMSE. The generalised nature of the maintenance model means that the conceptual model may be incomplete and so the OSRS for the ISMSE, derived from such a model is inevitably imprecise; more precise requirements can be formulated only after prototyping experiments have been carried out. The results of these experiments are fed back to allow a more complete conceptual model to be established: this is one of the aims of prototyping.

6.3.2.1.3 The objectives of the ISMSE

These determine the characteristics of the ISMSE: the *primary* objective of the ISMSE is to increase the productivity of the maintenance organisation, its *secondary* objective is to assist in researching the maintenance process.

Much of software maintenance concerns enhancements to existing software: the ISMSE will therefore possess some of the characteristics of a software development environment, but it

will offer complete support to the maintenance organisation for all phases of the software life cycle. It is intended that the ISMSE will be able to cope with the 'worst-case' scenario, that of implementing enhancements to an ageing, large, complex, unstructured software system, with one or more elements of the software configuration, (e.g. documentation), either missing or deemed to be unreliable. To achieve this, comprehensive support for the complete maintenance process must be offered, starting from the change request and culminating with the new release; providing the maintenance programmer with the means of coherently relating the separate pieces of information gathered by software tools, thus facilitating program understanding.

The ISMSE will *actively* aid the maintainer in deciding the best strategy for achieving an understanding of the relevant portion of the software system, prior to performing maintenance on that software. This is particularly important in the case of novice personnel.

The ISMSE will provide the means of unifying the separate activities which make up the maintenance process, increasing the effectiveness of automated tools through their disciplined application. In addition, the ISMSE will provide the means for easier monitoring of the maintenance process, serving as a tool for the administration of maintenance, i.e. it will support the handling of the 'paperwork'; information concerning the progress of the maintenance assignment, e.g. whether modules are under construction, completed, tested will be available.

6.3.2.1.3.1 Increasing the productivity of a maintenance organisation

The main objective of the ISMSE is the improvement of the productivity of the maintenance

organisation, by partially automating the maintenance process, and the provision of the means of *safeguarding* the gains in productivity, using a documentation paradigm, which is the subject of this thesis. In general terms, productivity can be defined as the quantity of work produced per unit time. A gain in productivity can mean that the time taken to complete a given task has been reduced, or that more work can be completed in a given time interval. In the context of a maintenance organisation, the definition becomes more difficult: it could be expressed as the number of lines of code maintained per unit time; or the number of requests for maintenance dealt with per unit time. There are many imponderables involved in these definitions, such as the difficulty in achieving an understanding of the program, and the human factor involved. The factors affecting productivity include:

1. Understanding of the software system prior to maintaining it.
2. Being able to obtain the required information to facilitate understanding.
3. Having a good set of easy-to-use tools to aid the maintenance process.
4. The facility for the component parts of the maintenance organisation to engage in meaningful dialogues - clarification of aspects of a change request can occupy a lot of time during a maintenance assignment.
5. Support offered for re-use, since the 'maintenance-history' contains both descriptions of problems and their solutions.

6.3.2.1.3.2 Research into the maintenance process

To continue in existence the ISMSE must be able to evolve in response to changes within

itself, e.g. the evaluation of *existing* tools may suggest ideas for new tools: and also within its environment, so it must provide support for the maintenance organisation to evolve, without the capacity to evolve, the ISMSE would become obsolete. Implicitly this means that the ISMSE must provide support for metrication when functioning as a test-bed for its own evaluation, the parameter of most interest being that of productivity, since as pointed out by Stenning [120] ...

> '...the role of an environment is to support the effective use of an effective process, the effectiveness in supporting a given process being measured in terms of convenience - the environment being the vehicle for achieving this convenience.'

6.3.2.1.4 The role of the ISMSE

The role of the ISMSE can be summarised as the provision of support for the maintenance process; and can best be described in the context of systems; in general terms a system is a collection of interrelated components which performs a function, i.e. it exhibits dynamic behaviour, observable in terms of its output, which characterises the system. In the context of computer science, a maintenance organisation is a system which performs the maintenance function; as such it is a self-regulating entity, interacting with its environment. It is both a deterministic and a probabilistic system; deterministic in the sense that the type and content of the information emerging from it will be predictable, to some extent, as a result of the input of data of an appropriate type, and probabilistic in the sense that the type of information input to the system can vary, and so the output can also vary.

Within a system, integration has a special part to play, since integration of the functions of the component parts of the system means that in the holistic sense the system is more than just the sum of its component parts. A system which exists as part of another system is usually described as a sub-system, with respect to the system in which it is contained; thus a sub-system can also be a system at the same time, if it contains a system within it. The *elements* of a system which are a prerequisite for its existence are its *environment* and its *boundaries*. In the context of computer science, the environment of an ISMSE is the data processing environment and the boundaries are the aspirations, goals and purposes, determined by its creator(s); these determine the scope of the ISMSE. The boundaries are obtained by modelling that part of the organisation which the ISMSE is to serve, i.e. its environment; this results in a model of the maintenance organisation. For such an organisation to be effective, there are three main prerequisites. The first is the understanding of the system's environment, i.e. the problem domain: Borgida et al [19] have shown that the majority of effort in deriving system requirements is spent in finding out and documenting knowledge about the environment in which the system is to operate.

The second is understanding the factors which govern the operation of the system, and how they are related. The third is the provision of adequate channels of information-exchange between its constituent parts, providing the means for communication and control. A system interacts with its environment by means of inputs and outputs, which are regulated by its boundaries, which act as filters for the system. The ISMSE is an open system, and has interfaces with its environment; this has implications for the requirements for the ISMSE, since its behaviour must be capable of adaption, in order to meet the changing demands

for information, which means that communication and control within the maintenance organisation are of vital importance. Changes within the organisation must be monitored, interpreted and evaluated, and appropriated responses devised: the ISMSE must provide support for this vital activity.

In chapter two it was pointed out that the generalised model of the maintenance process can not be viewed in isolation since it implicitly admits the existence of an organisation which will implement the model. The author envisaged a maintenance organisation whose structure is a hierarchical one, consisting of three main levels: managerial, supervisory and technical. In chapter two it was also pointed out that associated with each level of the organisation is a different type of information. The relationship existing between a particular level within the hierarchy of the maintenance organisation, and the type of information utilised by that level, deduced in chapter two, is reproduced below in Figure 6.6 for the benefit of the reader.

6.3.2.1.5 The Maintenance Organisation

To describe the various operations of a maintenance organisation, and the interrelationships existing between them, requires an abstraction of the organisation, so that an analysis of its information requirements can be undertaken, leading to decisions which will lead to an improvement in the performance, i.e. productivity of the organisation. The maintenance organisation is conceptually modelled as three interacting sub-systems, as shown in Figure 6.7 below.

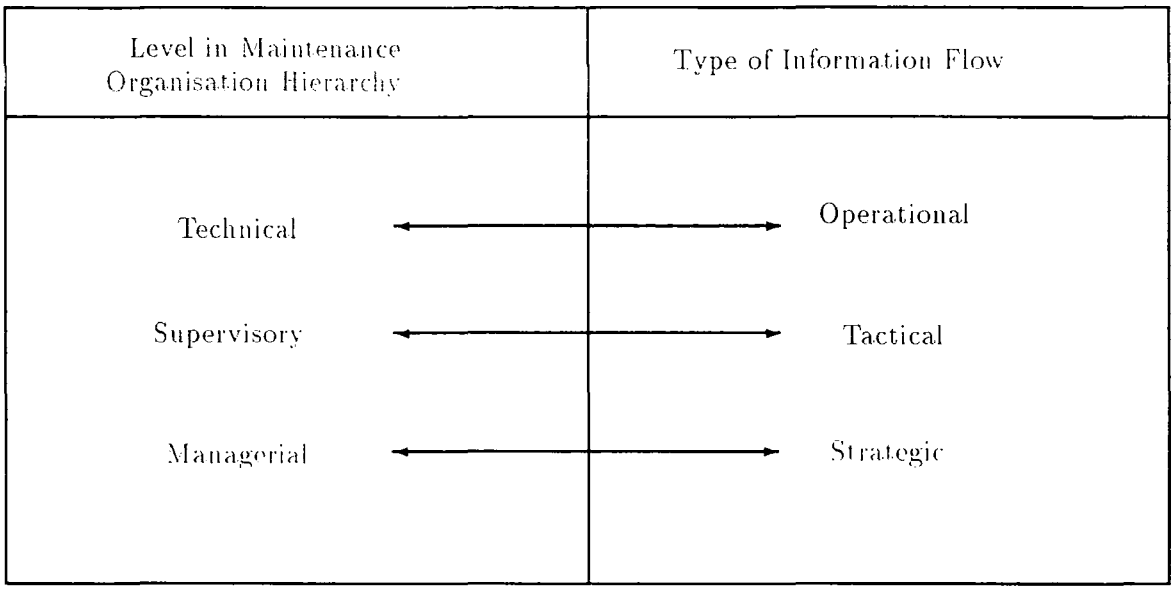


Figure 6.6: A Generic Maintenance Organisation Hierarchy and Associated Information Types

The interrelationships between these sub-systems is determined by the information needs of the maintenance organisation. The functions of these three sub-systems are described below.

Management sub-system

This corresponds to the managerial level of the maintenance organisation, and consists of people and activities related to the planning, controlling and decision-making aspects of the operations sub-system.

Operations sub-system

This corresponds to the supervisory and technical levels, and consists of activities, information flow, and people directly related to performing the primary function of the organisation, i.e. the processing of a change request for maintenance.

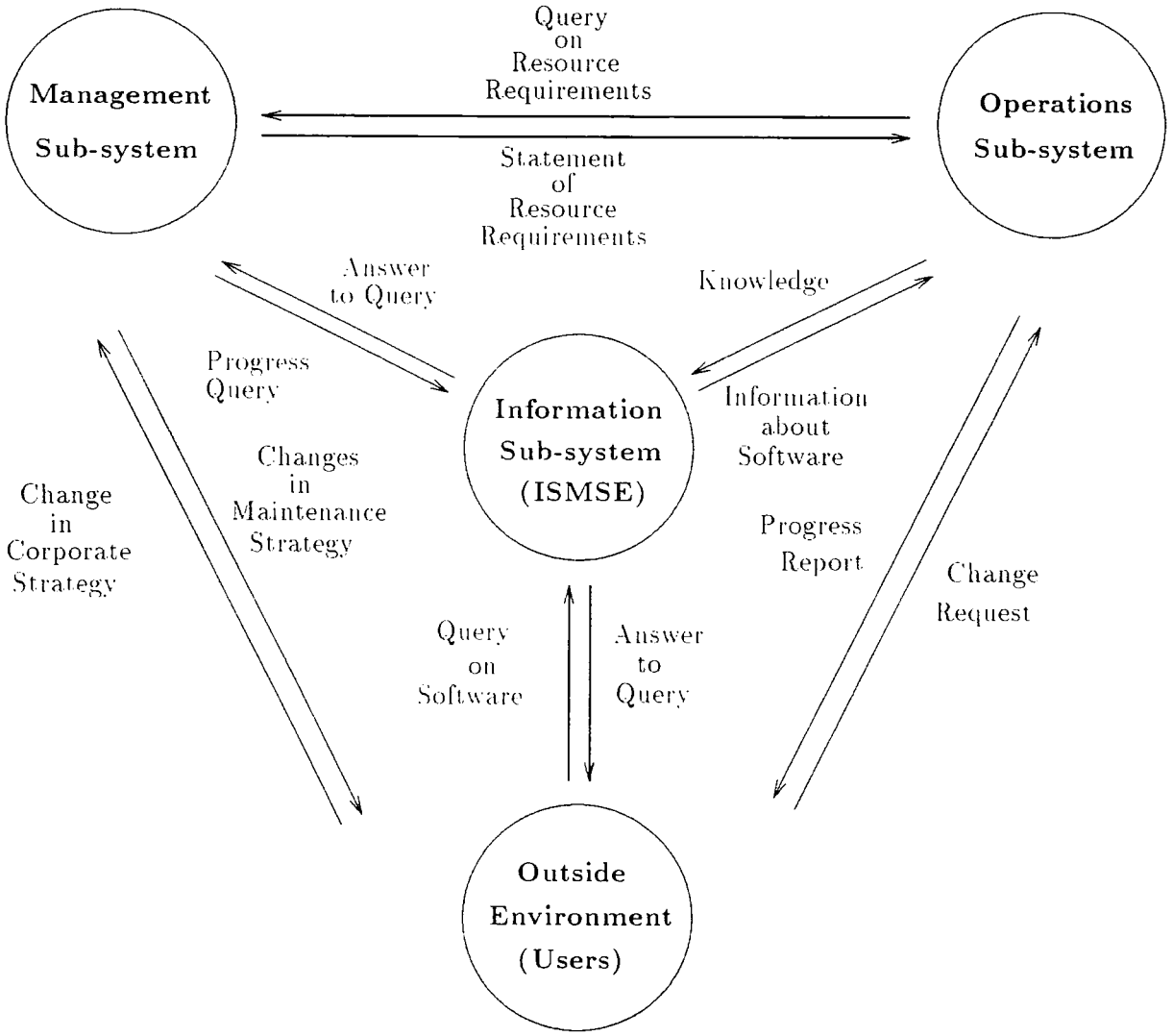


Figure 6.7: Conceptual View of a Maintenance Organisation

Information sub-system

With respect to the maintenance organisation, the ISMSE is a sub-system which itself contains sub-systems, e.g. database, user-interface. The common entity exchanged within the ISMSE, between these sub-systems is information. It is important to understand the nature of the ISMSE's sub-systems and the information flows which enable its components to perform synchronously. The information sub-system receives data from the operations and management sub-systems of the maintenance organisation and from the outside environment, i.e. *users* of the software, the maintenance of which is the responsibility of the maintenance organisation. These inputs are transformed into some meaningful information by the operations sub-system, using the ISMSE, which is then transmitted to the appropriate sub-system. The purpose of the information sub-system is to satisfy the information requirements of the maintenance organisation, of which it is a part and to make possible meaningful dialogues between the other sub-systems. This role-abstraction serves to underline the fact that the ISMSE is a tool, which interacts with its environment, and must be capable of adapting to a *changing* environment, otherwise it will become obsolete, i.e. the ISMSE must at all times be *complementary* to the maintenance organisation.

6.3.3 The OSRS document

The OSRS can be partitioned into two classes: functional requirements, which state the actual functions which the system must implement, and non-functional requirements, which express practical constraints such as performance specifications and memory requirements.

Since prototyping is to be used to refine the functional requirements definition, and prototyping is not concerned with performance constraints, this aspect of the requirements for an ISMSE will not be pursued here. The requirements given here constitute the highest level in the hierarchy of the IEEE OSRS document, and as such constitute a slice through the hierarchy, since the complete requirements for a software system as complex as the ISMSE are beyond the scope of this thesis. The format of the requirements definition for the ISMSE is shown below in Figure 6.8.

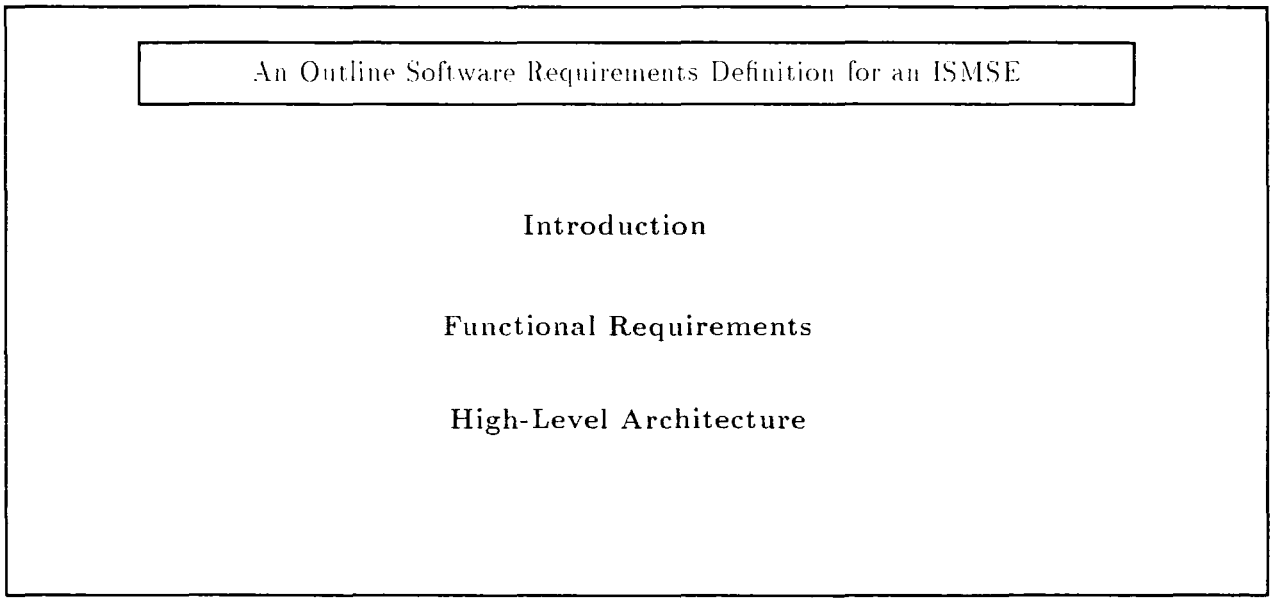


Figure 6.8: Requirements Definition for an ISMSE

6.3.3.1 Introduction

Software maintenance needs to be placed on a firm foundation, in the same way as software development was in the 1970s, through the adoption of a disciplined engineering approach. Other research initiatives aimed at partially automating the maintenance process by the

use of software tools have concentrated on particular aspects of the maintenance process, e.g. [48]; none have attempted to improve the approach to maintenance in a comprehensive manner.

6.3.3.2 Functional Requirements

6.3.3.2.1 Overview of functional requirements for an ISMSE

From the conceptual model, i.e. the highest level of abstraction, the requirements can be stated as the extraction of information from software, and the management of this information, to increase the understanding of the software.

The maintenance model, derived in chapter two is the product of the analysis phase, and the high-level requirements for the ISMSE are based on the model, which is reproduced below for the benefit of the reader:

1. Verification of need to modify the software, i.e. the program *and* its associated documentation.
2. Understand the software.
3. Modify the software, including documentation.
4. Validation of the software (i.e. the functional specifications) and regression testing.

The maintenance model expresses the highest-level requirements of the ISMSE, the analysis of the objectives of the ISMSE results in the refinement of this model, giving the conceptual

model. Provision of support for the actions of the ISMSE is through the use of software tools to extract information from a software system, and to provide the means of managing this information, i.e. a software engineering database, to increase the understanding of a software system. The ISMSE will therefore provide:

1. Basic run-time support facilities for the phases of maintenance, giving continuous and effective support for maintenance, addressing all aspects of software maintenance and offering complete support to the maintenance organisation for all the phases of the software life cycle, particularly for the understanding phase of the maintenance process.

2. Effective management and control through the provision of support for the administration of maintenance, particularly the storage of information produced by the ISMSE, concerning the progress of the maintenance assignment, and the subsequent use of that information, including the ability to ascertain the current position with regard to, e.g. whether modules are under construction, completed, or tested.

3. The provision of a set of tools covering the whole software life-cycle

4. Support for the integration of tools.

   Many traditional software engineering environments comprise an operating system and filing system, together with an ad hoc collection of tools to cover some parts of the maintenance process. Individual tools are used to automate at least some part of the maintenance process, the level of support provided by a tool is a function of the degree of sophistication of the environment hosting the tool, e.g. the APSE contains rules which a tool makes use of to check information concerning a software system under

development or maintenance. Since the ISMSE's architecture will be open, it will be able to integrate third-party tools including those of tool vendors.

5. The ability to re-use software components, e.g. programming cliches.

6. Support for multilanguage developments.

The functional requirements are now discussed in more detail.

6.3.3.2.2 Requirements for object base

Introduction

The object base is the nucleus of the environment, its primary role being that of a repository of information concerning the software that is being maintained, its secondary role is to act as an integration mechanism for the environment.

Requirements for information repository.

The database will allow the maintainer to perform queries and updates on stored items of information, using e.g. a relational query language as a tool to provide the maintainer with different views of a software system. The database must be typed, i.e. it must have knowledge of certain properties of the objects that it contains, and can prevent their misuse by incorrect tools. This simplifies tool development since it provides the tool writer with an appropriate framework: this approach contrasts with, e.g. some programming environments where no central structure is imposed and the interface management varies according to

which tool is being used. Each object in the database will be date and time stamped with a locking mechanism to prevent more than one user from accessing an object at the same time.

Requirements for integration mechanism

The ISMSE must integrate the tools that are associated with the three phases of the maintenance process. In general terms the need for integration occurs whenever two or more interrelated components have a common interface [55]. Identification of the interfaces present in the ISMSE will indicate where integration is required. The obvious interfaces are the user interface and the tool-tool interface. The author defines integration in the context of this research topic as 'The incorporation of tools into a coherent unit for the capture, generation and management of information concerning a software system.' The desirable properties bestowed upon the environment as a result of integration include:

1. Synergic sequential invocation of tools, with concomitant smooth transmission of function, so that the repository of information concerning the system under investigation (e.g. structure charts, symbol tables), can be managed by the coordination of tools such as cross-referencers, restructurers.

2. The ability to retrieve information from the database and present it to the maintainer, in a way specified by the maintainer, i.e. provide the maintainer with different views of the software.

3. Flexibility and extensibility.

The overall effect of an integration mechanism will be to serve as a unifying influence on the environment; such that it becomes the embodiment of a number of closely-linked interrelated functions resulting in its manifestation as a single tool.

It has been observed by Glass [46] that some software tools produce information that could prove to be of considerable value to the maintainer, but these tools do not output this information; examples of such tools are the compiler and the restructurer. The integration mechanism should therefore offer an interface to these tools.

6.3.3.2.3 Requirements for toolset

6.3.3.2.3.1 Introduction

It is envisaged that the ISMSE's tools will have a 'knowledge' of the functions performed by each other and also of the information processed and produced by each other. This knowledge includes:

1. the transformations performed by tools on objects in the database
2. the format of the data required by other tools
3. how a tool is invoked
4. values held in variables, constants, via parameters

The toolset will be capable of providing support for all phases of maintenance, but will be biased towards program understanding; tools for verification and validation of programs will also be included. The toolset will be minimised by examination of tool-function to determine, e.g. whether any useful information is generated by a tool, but is not output. The criteria for the selection of the toolset will need to be established, using the adopted maintenance model. Knowledge of the *types* of tools to be used will aid in drawing up the requirements for the proposed ISMSE: most commercially-available tools combine many functions into one tool.

6.3.3.2.3.2 Problems with tools in available support environments

Existing software environments have encountered problems with regard to the tools they contain. Donahoo [36] summarises these problems:

1. Tools lack uniformity, completeness and compatibility.
2. No system has a complete set of tools.
3. Many tools are system-dependent, and language-dependent - even tools designed for a given system and a given language may be incompatible.

These problems underline the need for a complete set of tools which *are* compatible. However, the amount of output the maintainer is faced with increases proportionately with the number of tools being used: the reduction of the number of tools to a minimum requires that strict criteria for tool selection be drawn up. One approach to reducing the amount of information is to make tools function interactively, another approach is intertool communication, one tool

accepting the output of another as its input, as Unix tools do. Another important aspect of tool usage is how best to *present* the captured information to the maintainer: this is covered in detail in chapter 7.

6.3.3.2.4 Requirements for user-interface

Introduction

These requirements embrace the non-functional aspects of the ISMSE. The ISMSE is an 'in-house' tool for an organisation and the users outside of the ISMSE use both the ISMSE and its products. As well as their obvious ability to perform arithmetic, computers facilitate communication and control. The diverse user-community makes necessary the construction of a view mechanism, so that each category of user has access to that subset of the data which is appropriate to his needs, and also prevent access from the user which may affect the integrity of the ISMSE and its facilities. Users can also use the ISMSE for retraining purposes, when enhancements are made to software; for this reason the ISMSE should include facilities for Computer Assisted Learning (CAL). The user interface will need to be enhanced, in relation with current environment interfaces, to cope with the diverse community who will use the ISMSE.

The user interface should:

1. be a WIMPS interface, independent of any host machine.

2. make it possible for tools to be invoked individually, and/or serially. When tools are used interactively, the user must be able to exert control over the tool. It must also be possible to use tools in batch mode. It must not be possible to misuse a tool.

3. allow access to the host operating system, e.g. for initial connection to the ISMSE, and for access to any tools which are part of the host operating system, using the command language of the host operating system.

4. permit the suspension or termination of the current function or program, and the return to the command language interpreter, resuming the current function or program

6.3.3.2.5 Design of user-interface

User interface design is concerned with the interaction of humans and complex computer systems. The main function of the user-interface is to reduce the cognitive overhead associated with complex tasks, making possible the abstraction from task details, allowing the user to concentrate on high-level issues, therefore increasing his productivity. Integration of an ISE such as the ISMSE can be achieved at the user-interface: the design of the user-interface combining the disciplines of Computer Science and Psychology. The main dilemma is the decision as to whether to base the interface on a command language, which is powerful and complex, suited to expert users, but difficult to learn, or to base the interface on a system of hierarchical menus, suited to novice users, which an expert may find tedious and cumbersome to use. Hierarchical menus could be used to operate the windows part of the environment, with a command language operating inside those windows.

6.3.3.3 A High-level Architecture for an ISMSE.

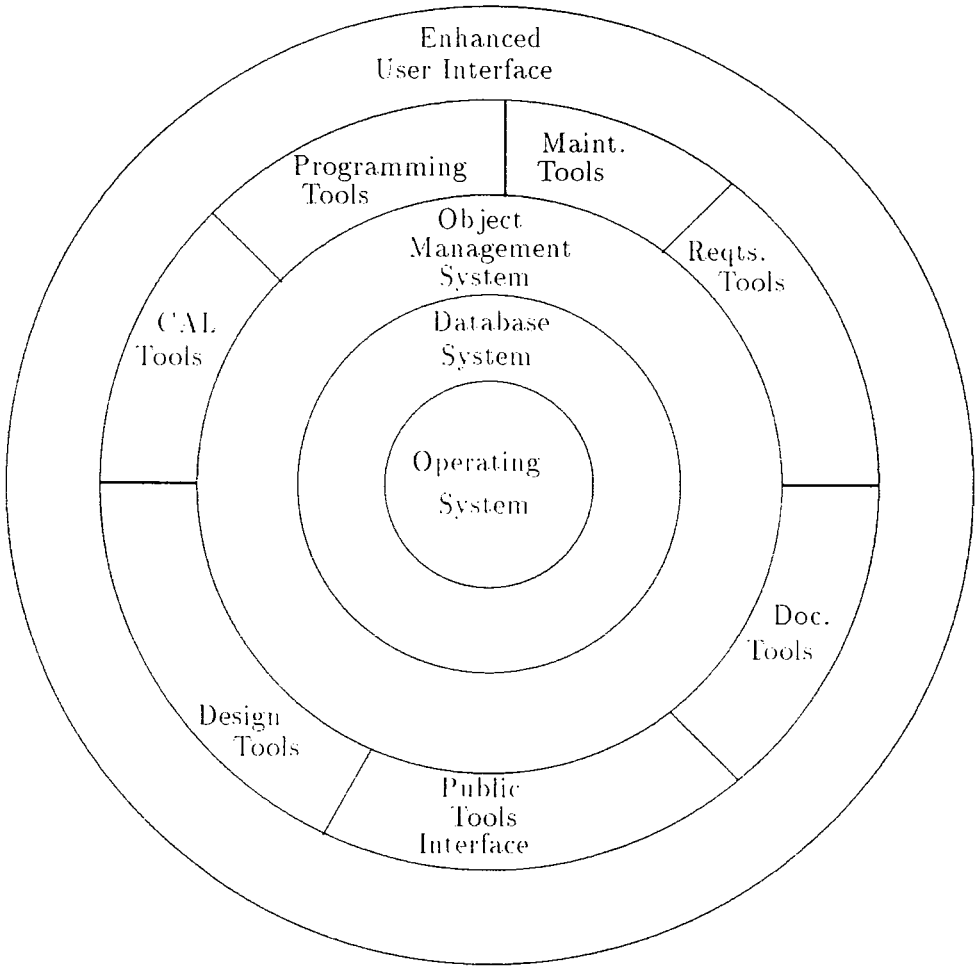
The high-level design of the ISMSE comprises its observable external characteristics, e.g. the user-interface, and its high-level architecture, i.e. the structure of the ISMSE. The components of the ISMSE and the way they relate to each other describe its architecture, which will be a layered one, as shown in Fig 6.9 below, to simplify the design and maximise application program independence. The maintenance environment must be flexible and extensible, i.e. it must possess an open architecture so that the tools belonging to the host operating system, and third-party tools can be incorporated into the environment, and so that any development method can be supported. The ISMSE consists of a collection of interconnected modules, the design of the ISMSE is concerned with the interaction of these modules, due to their configuration and structure. The modules are:

1. input and input modules

   These correspond to the user-interface of the ISMSE. The input module is concerned with the handling of input data from tools, or from the maintenance organisation, or from users; the output module is concerned with the type and volume of information required from the information system, i.e. views of software

2. control and procedures modules

   These correspond to the toolset of the ISMSE and are concerned with how data and information are handled, from input stage through processing to output. Tools convert data into information through tool-tool interaction.



Key	
Doc.	Documentation
Maint.	Maintenance
Reqmts.	Requirements

Figure 6.9: A High-Level Design for an ISMSE

3. data repository module

This corresponds to the object base of the ISMSE.

6.3.3.3.1 Design and prototyping

The design method adopted is that of rapid prototyping, to highlight any weaknesses in the requirements definition, early in the development of the software. Prototyping may be used as a means of learning about a problem domain or as a means of incremental development of a software system, as indicated below in Figure 6.10.

Prototyping is a process that enables a developer to create a model of the software that

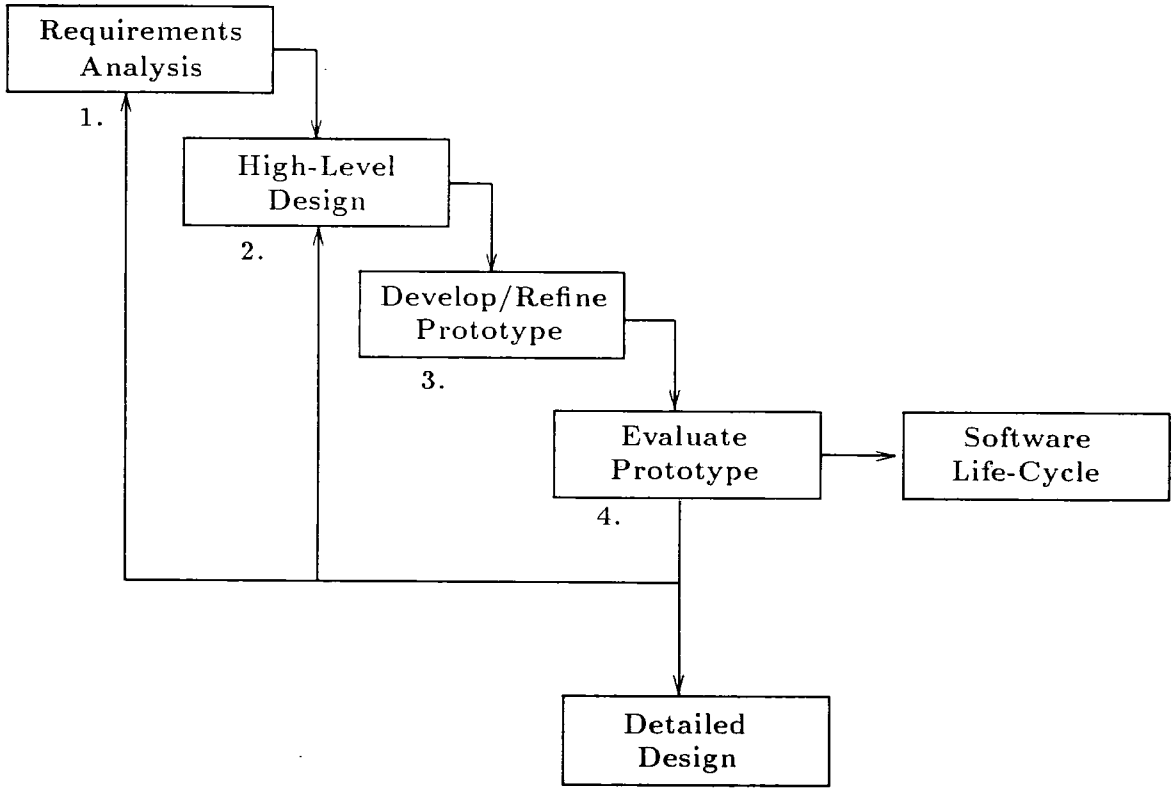


Figure 6.10: Prototype Life-Cycle Model

must be built. The model or prototype may be designed to depict one or more aspects of

e.g. a human-machine interface, or a set of functional or performance requirements that are questionable. The aim is to construct a working prototype, which constitutes a subset of the function of the ISMSE, i.e. automated support for the documentation paradigm. The sequence of events for the prototyping paradigm is:

1. identify whatever requirements are known and outline areas whose further definition is mandatory

2. 'quick' design is done leading to the construction of a prototype: this focuses on a representation of those aspects of the software which will be visible to the user, e.g. user-interface.

3. evaluation of prototype is used to refine the requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the customers' requirements. Ideally the prototype serves as a mechanism for precisely identifying the software requirements.

Evaluation of the prototype leads to a detailed design specification, describing control flow, data representation and other algorithmic details, within the modules.

6.4 Summary

A design process has been proposed for an ISMSE, highlighting the relationship between requirements and design and describing the role that abstraction plays in obtaining a high-level requirements definition for an ISMSE.

The purpose of the software requirements specification has been described and a method given for expressing and gathering the SRS. The requirements definition was developed by examining the objectives of the ISMSE and using a description of its role to produce a conceptual model.

The functional requirements for the components of the ISMSE have been described and a high-level architecture suggested: together with a strategy for validating the requirements definition, using prototyping.

Chapter 7

An Information Structure for an ISMSE

7.1 Information Capture and Processing

The crucial aspect of improving the productivity of a maintenance organisation is the reduction in the time needed to gain an understanding of a software system, through partial automation of this aspect of the maintenance process. In order to achieve this aim there are two vital requirements: the capture and processing of information to further understanding,

using an integrated set of suitable software tools, and good quality redocumentation, to preserve the maintainability of a software system, (described in chapter two) so that future maintainers can achieve an understanding of the software more easily. Both these requirements can be satisfied by providing the necessary mechanisms for the capture, management, analysis and subsequent archival of information generated during the course of a maintenance assignment, using a documentation paradigm which provides facilities for abstraction. The task of satisfying the above requirements can be decomposed into the following subtasks:

1. What information to capture.
2. Choice of information structure used to store the captured information.
3. Where to put the captured information within the information structure.

7.2 What information to capture

Typical of the information needed by a maintainer which is captured during the course of a maintenance assignment and stored in the database is shown below:

1. Which software is currently being maintained
2. A record of the change-request
3. Whether problem reports are on file

4. Requirements for change to software

5. Specifications of changes to software

6. Design documentation

7. Program source code

8. Program documentation

9. Test data

10. Results of analysis of a database object (cross-reference listings, call graphs etc.), i.e. views of the software being maintained.

11. Objects produced by transformation tools, e.g. structured code from unstructured code

12. The language in which program code is written

13. The *version* of the language in which the program code is written

There is a need for a toolset to automate partially the information-capture and subsequent storage in the environment database. Software tools and the associated technology are currently available from vendors; however, introducing tools into software engineering environments has attendant support problems, summarised by Donahoo et al [36] and listed below.

1. A single tool or technique is insufficient; a combination of consistent and complementary tools should be selected.

2. Automated tools must be supported with sound management, and organisational concepts and procedures.

3. Tools require the interaction of human experience and judgement and can only assist the user, not replace him.

4. Tools must be reviewed periodically for enhancement, utilisation of new technologies, or retirement.

The generalised maintenance model (derived in chapter two) has made possible the selection of a combination of consistent and complementary tools, shown in Figure 7.1 below, and its associated conceptual view of a maintenance organisation, (derived in chapter six) provides the framework for sound management and organisational concepts and procedures, required for the support of automated tools. This chapter provides a documentation paradigm to be used within this framework, to maximise the effectiveness of software tools, particularly with regard to program understanding, providing an information structure to support the interaction of human experience and judgement. Finally, the documentation paradigm aims to provide a basis for the archival of information, including that concerning the version of a software tool being used, which will enable the performance of a tool to be monitored and evaluated, to help in deciding whether a tool needs to be retired or enhanced.

7.2.1 Analysis of tool classes and tool functions for information capture and processing

It is apparent that some tools produce much data and little information, and the maintainer is again faced with a manual task of scanning large amounts of printed matter: highlighting again the need for **abstraction**. A database query facility can provide help for abstraction and so is important in the understanding phase of the maintenance model, as well as in the verifying phase, but cannot be used until other program-understanding tools have been used to put information *into* the database. The database query facility can extract views of software which can lead to an increased understanding, in turn this can result in new information or knowledge being added to the database. The United States National Bureau of Standards [90] has provided a taxonomy of tool-types which recognises three classes of software maintenance tools.

1. Transformation tools.
2. Static analysis tools.
3. Dynamic analysis tools.

The environment will utilise these tool classes, using the generic tool types which are enumerated for each phase of maintenance, as shown in Figure 7.1 below. A short description of these generic tool types now follows, to illustrate the complementary nature of the toolset chosen to support the maintenance model, and to show the different types of information produced by these tools.

Phase of Maintenance	Generic Tool Type
Verifying request	Database Query (T)
Understanding	Cross-referencer (S) Structure-charter (S) Reformatter (T) Documentor (T) Restructurer (T) Reverse Engineering (T) Database Query (T) Execution flow tracer (D)
Modifying	Cross-referencer (S) Structure-charter (S) Execution flow tracer (D) Documentor (T) Editor (T) Translator (T)
Revalidation	Cross-referencer (S) Test data generator (D) File comparator (S) Debugger (D)

Key	
T	Transformation Tool
S	Static Analysis Tool
D	Dynamic Analysis Tool

Figure 7.1: Generic Tool Types to support the Maintenance Model

7.2.1.1 Transformation tools

These tools operate on strings of input producing modified output; some examples are given below.

1. Database Query

   The main function of this tool is to enable a maintainer to increase his understanding of the software by providing views of the software and converting information into knowledge, discussed in chapter three. The kind of tool which can act as a query facility is dependent upon the conceptual schema chosen for the data structure, which is deposited in the database.

2. Reformatter

   This tool is sometimes known as a prettyprinter and operates on a file of source code with a view to enhancing its readability, and therefore understanding, by:

   (a) sequencing statement numbers

   (b) indenting statements

3. Documentor

   The aim of this tool aims to make possible changes to all the different kinds of documentation contained within a large software system, both textual and graphical.

4. Restructurer

   This tool takes as input unstructured source code and as output produces structured source code, which is logically equivalent to the original.

5. Reverse Engineering

   This tool aims to analyse source code to capture design information.

6. Editor

   The main function of the editor is to make changes to the source code.

7. Translator

   Examples of its use include:

   (a) Language converter.

   Converts one language to another.

   (b) File converter.

   Converts one file format to another.

7.2.1.2 Static analysis tools

These provide information about a program, without actually running the program: some examples are given below.

1. Cross-referencer

   This tool reveals logical relationships between entities within a program or between programs. Cross-references for variables or of calls, help to establish the structure of a program or sub-routine. Cross-referencers can be used in debugging and impact analysis, and can be interfaced with:

   (a) Documentation tools to provide automatic documentation of source code.

(b) Graphical tools to give a pictorial representation of the program.

2. Structure Charter

   This tool graphically produces the relationships between the various components of the software, at the level of procedures and functions.

3. File Comparator

   Supplies instances of similarities and differences between text or data files and assists in version control and maintenance of source code. File comparators can assist in comparing the actual output of programs with the idealised output of the program when functioning as laid down in the functional specification - this is an important tool for verification and validation.

7.2.1.3 Dynamic analysis tools

These provide information about how a program executes, and are of help in detecting errors, some examples are given below.

1. Execution Flow Tracer

   This shows the sequence of actions carried out on a statement by statement basis, or just those statements which alter the flow of control, or which change the value of a specific variable.

2. Debugger

   This tool enables the maintainer to step through the program, at a chosen pace, mon-

itoring values held in variables and evaluating boolean conditions.

In most software engineering environments the operand for a software tool is the file; different tools have different requirements as regards file formats and the output of one tool is often the operand for another tool, i.e. these software tools produce objects of different types, highlighting the need for tool integration.

7.3 Choice of information structure used to store the captured information

The information structure to be used as a basis for the documentation paradigm must be able to store the diverse types of information, both textual and graphical, produced by the generic tool-types described earlier, and also provide the means of organising this information prior to converting it into knowledge. The form of the maintenance log to be kept by a maintenance organisation is determined by an examination of the support provided by an ISMSE, described in chapter six. The acquisition of knowledge is facilitated by the organisation and management of information, the processing of this information into knowledge being achieved by human intervention. The role of the maintainer is to act as inference engine, producing information, then knowledge from raw data, and entering this knowledge into the object base, as shown in Figure 7.2 below. The knowledge, and the ensuing understanding stemming from the generated information, need to be documented.

and an information structure is required that can also provide support for this activity.

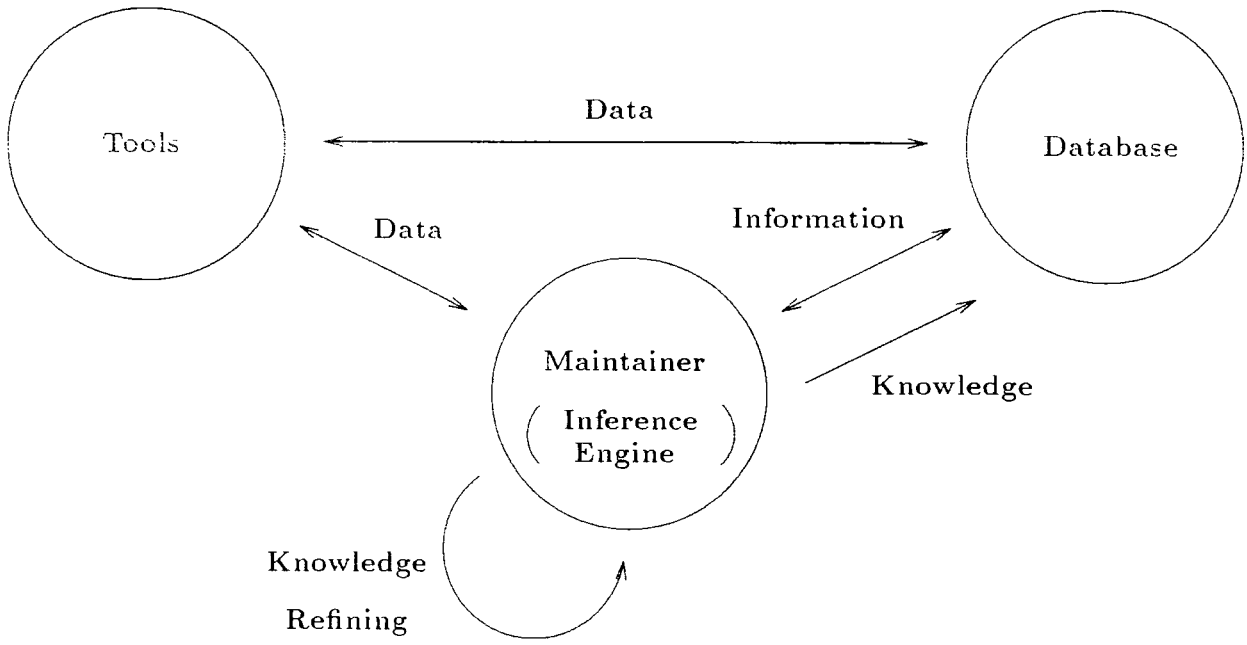


Figure 7.2: The role of the Maintainer - schematic

The recognition of the importance of the part played by abstraction in the domain of program understanding resulted in the choice of a hierarchical information structure for the maintenance log. Items of information describing a software system entered by the maintainer, or produced by a software tool, may be linked and stored as a book format, shown below in Figure 7.3. The literature reveals other initiatives concerning the 'book paradigm', in the domain of software engineering, but of these only one is concerned with the maintenance of software, that of Oman and Cook [94]; the remainder are concerned with software development. Oman and Cook [94] proposed a method of formatting programs consistent with programmer comprehension strategies and maintenance activities, i.e. the book format is only used as a code-viewing paradigm. McKissick and Price [82] proposed the use of the 'book paradigm' in connection with the progress of software development activities. This

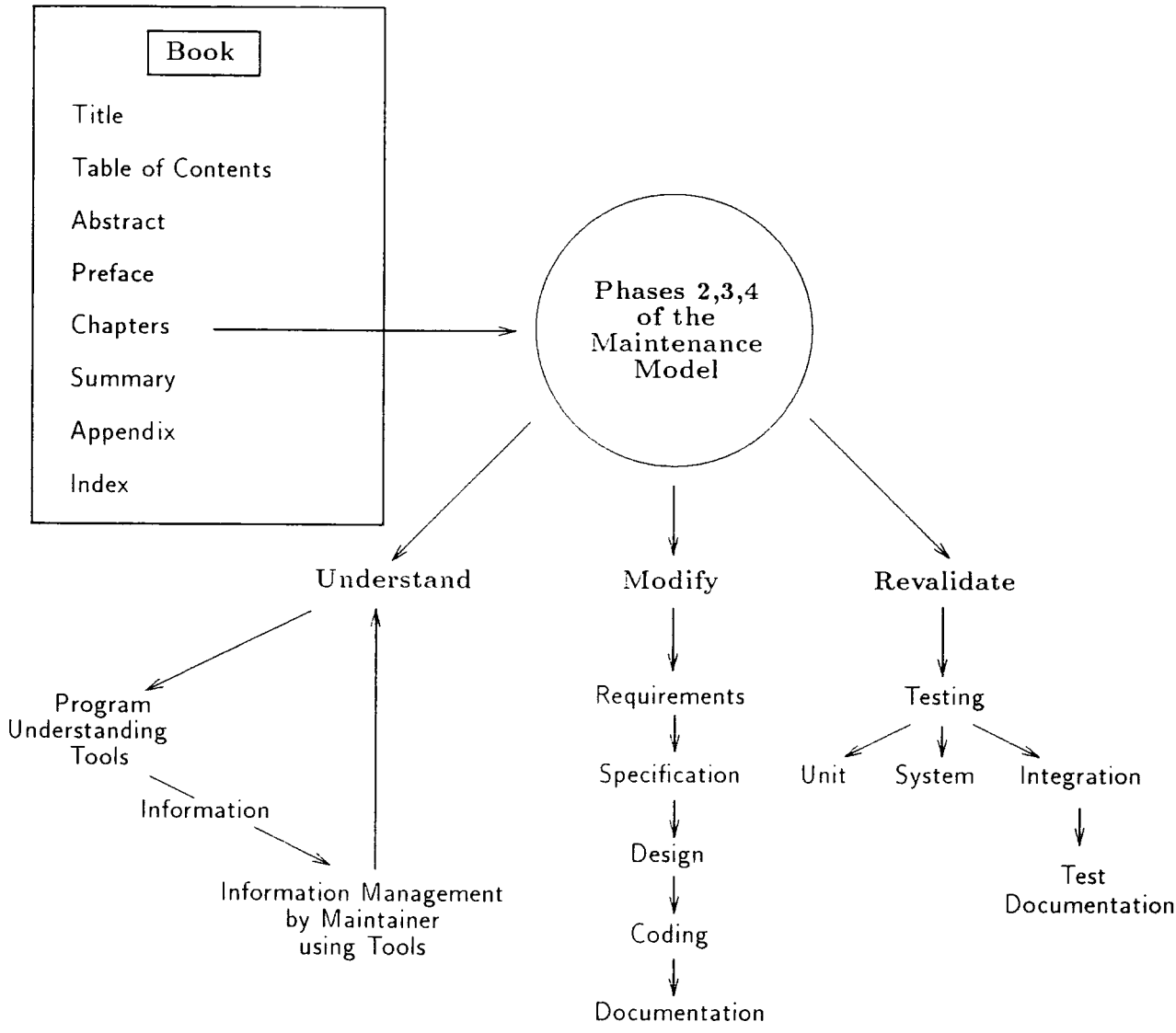


Figure 7.3: The Book Format as a Data Model for the Organisation of Information Concerning a Software System

was a paper-based system, and did not develop the potential of the 'book paradigm' beyond that of the keeping of a simple notebook. Kempe [62] proposed the use of a 'book paradigm' for developing a data management kernel for the management of structured documents, concerning an object management system based on *hypertext* principles. The approach in this thesis is not based on hypertext principles. Koenig [63] proposed the use of the 'book paradigm' in connection with software development activities, and uses an enhanced electronic version of the software development notebook idea proposed by McKissick [82]. The use of the 'book paradigm' in this thesis is concerned with the *maintenance* of software, and proposes a much wider application of the 'book paradigm' than other initiatives.

7.4 A Maintenance History for a Software System

An anthology of hierarchical maintenance logs, i.e. books, comprises the 'Maintenance History' of the software and forms the basis of a documentation paradigm for the ISMSE, the aim of this thesis. Each book of the anthology encompasses a version of the software system. Each time a maintenance assignment is carried out it should be written up in book-form; the 'book' will contain a data dictionary, holding information about variables, constants, and routines - available from a cross-referencer.

7.5 The Maintenance History as an ADT

The work earlier in the chapter described the tools which are used to capture information during software maintenance, and gave an informal description of the structure used to store this information, leading to the establishment of a Maintenance History for a software system. One of the aims of the work in this chapter is to describe this information structure in a more formal way, as an **abstract data type**, which will be referred to as the ADT Maintenance_History.

In computer science the term abstract data type or data abstraction is usually associated with a computer program, and only exists during the time of the invocation of a program, able only to access objects at the file level. The information structure described in this chapter is actually a **database view** which is a persistent object and must exist beyond the invocation of a program which manipulates it: nevertheless it can be regarded as a **data abstraction** or an abstract data type, since this permits the definition of the view, as described below and implicitly limits the update operations performed by tools, which are allowed on the view. The information contained in a database is accessed and updated using a Data Base Management System (DBMS). The DBMS provides independence between the physical representation of data and the user's view of it [124], this is made possible through the design of a three level architecture for a database, described below.

7.5.1 Database Management System Architecture

7.5.1.1 Introduction

A data view specified in a language which the database management system software can understand is known as a **schema**. The database management system architecture has three levels, consisting of three related schemas viz: the external schema, the conceptual schema, and the internal schema, as shown in Figure 7.4 below.

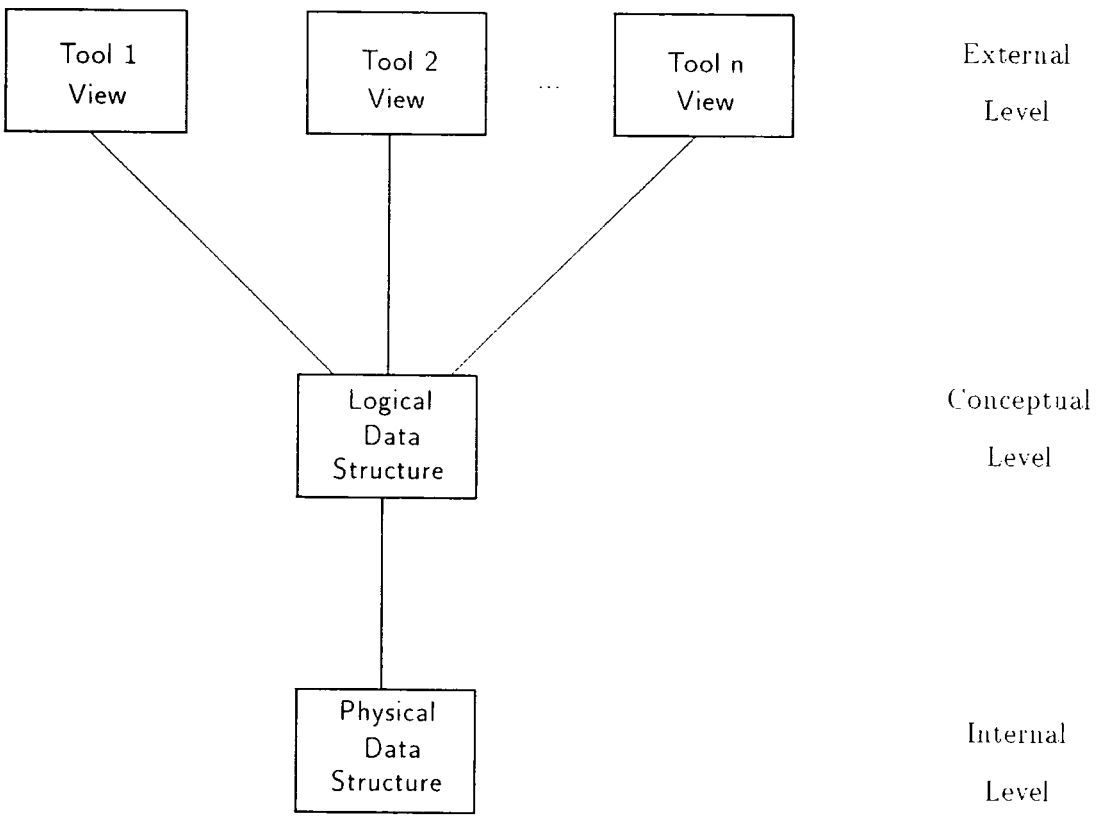


Figure 7.4: The ANSI/SPARC DBMS Three-Level Architecture.

7.5.1.2 External schema

The information structure comprising the Maintenance History is to be held in a database, and since no one view of a software system is sufficient to permit effective software maintenance, maintainers need to extract different views of software, by using different software tools. This is made possible using an external schema which provides one or more programs e.g. a software maintenance tool, with a local view, which can be derived from the conceptual schema. Programs requiring identical local views may share the same external schema. Properties of the data, such as the format of the data items or the sequence in which data is seen, may be specified by an external schema, but it cannot override any of the constraints imposed by the conceptual schema.

7.5.1.3 Conceptual schema

Formally, a conceptual schema is a neutral integrated view of a data resource and acts as a bridge between the internal and external schemas as described by the ANSI/SPARC three-level database architecture.

Informally, a conceptual schema is a description of the data structure of interest to the maintenance organisation which is to be stored in the database. The schema must be comprehensive, since database management systems differ in their degree of comprehensiveness, and ease of use of their conceptual schema facilities.

The conceptual schema is of great importance and performs a vital role in that it:

1. specifies the logical data content of the database, i.e. the Maintenance History, whose data structure forms the basis of the documentation paradigm for the ISMSE, and determines the constraints which apply to this data structure, i.e. the update operations allowed on it. This largely determines the scope of the ISMSE and directly reflects the level of tailoring to a particular process, i.e. the maintenance model, derived in chapter two.

2. provides the basis for integration, since the various tools and facilities must all operate to a single common data structure as defined by the conceptual schema.

The documentation paradigm serves as an initial conceptual schema for the environment database and is a portable information sub-system for the management of information concerning the maintenance of software. The conceptual schema is stored as a graph, and determines the structure of the database, establishing fixed rules regarding the way information is held in the database.

The documentation paradigm was derived by examining the information requirements for the information sub-system and the applications, i.e. the environment's toolset, which accesses data, and updates the data held there. The sub-system description is given by the maintenance model used within the framework of a maintenance organisation.

The Maintenance History describes data controlled by the maintenance organisation, and is extensible and consistent, enabling the data resource to evolve, functioning as an integration mechanism, by absorbing disparate pieces of information concerning the software system, making them part of a coherent information structure.

The evolution of the maintenance organisation, means that the nature of the information used by that organisation will also evolve and so the conceptual schema describing that information will have to be changed. If the conceptual schema is an accurate model of the organisation, then any change seen by the organisation as being a simple change should imply a similar change in the conceptual schema, and vice-versa. The conceptual schema can be regarded as a relatively stable, long–term view of the data, which is capable of evolving with the organisation.

7.5.1.4 Internal schema

This describes how the conceptual schema is physically implemented: at the level of stored records, stored record formats, indexes, hashing algorithms, pointers, block sizes for consistency (e.g. that each external schema is capable of being derived from the conceptual schema) and must use the information in the schemas to map between external schema and the internal schema via the conceptual schema.

7.5.1.5 Summary of Database Architecture

Individual tools or groups of tools may have their own local view of the conceptual schema, i.e. the output from the tools which are used to gather information concerning the software and are used to update the information held in the database must conform to this data structure, which maps to the conceptual schema. These views may provide mappings between the data structure within the database and the data structure within the tool; but

these different views must all be compatible with the conceptual schema. The conceptual schema is the hub of the database architecture. Each external schema provides one or more programs with a local view which can be derived from the conceptual schema; the internal schema describes how the conceptual schema is physically implemented.

7.5.2 Description of the Conceptual Schema as an ADT

The conceptual schema for the environment database is expressed as an abstract data type for the following reasons:

1. Abstract data type is a fundamental and unifying concept in computer science, enabling the separation of the 'what' from the 'how'; which enables implementation decisions to be postponed as late as possible. ADTs are used in almost all stages of software development, particularly in specification, design and implementation.

2. The separation of the specification of a piece of software from its design and subsequent implementation - specifying software in terms of the data to be processed and the operations that must be performed on it, i.e. as an ADT, has proved to be an excellent way of producing reliable software and of reducing costs.

3. The idea of an ADT can be used in program development in conjunction with established techniques, such as stepwise refinement.

An ADT is an abstract model of a problem domain, or part of a problem domain, considered and defined purely in terms of the sets of values that variables of the type may take, and is bound to a set of operations which may be performed on the type *without* consideration as to the implementation of these operations. An ADT attempts to model as closely as possible the problem domain and is a concept which is usually associated with computer science, but actually predates it. Strictly, an ADT is a triple (D,M,A), consisting of a set of domains D, a set of operations M, each with range and domain in D, and a set of axioms A, which together specify the properties of the operations in M. By distinguishing one of the domains d in D, a precise characterisation is obtained of the data structure that the ADT imposes on d. As an example the natural numbers comprise an ADT, whose domain is:

$$0, 1, 2, \ldots$$

.. and there is an auxiliary domain ..

$$TRUE, FALSE$$

The operations on the type are ZERO, ISZERO, SUCC, and ADD and the relationships between these operations are specified by the axioms below:

$$ISZERO(0) = TRUE$$

$$ISZERO(SUCC(x)) = FALSE$$

$$ADD(0, y) = y$$

$$ADD(SUCC(x), y) = SUCC(ADD(x, y))$$

These axioms comprise a precise specification of the semantics of the ADT known as the set of natural numbers. Without the associated operations the sequence of symbols

0, 1, 2, ...

has no meaning, since by themselves they provide no semantic description of the abstract data type.

There are problem domains which require the use of more complex abstract data types, than that illustrated in the previous example. These can be constructed from simpler ones, which allows the definition of a hierarchy of ADTs, the highest being the nearest to the problem domain; note that this also implicitly means the creation of new operators to manipulate the new ADTs, this is known as **procedural abstraction**. The key role that computer science has played concerning ADTs is the **enforcement** of the type, i.e. the type can only be manipulated using the operations provided, this is achieved through the concept of **information hiding**. This means that in the context of software development a program can be split up into separate tasks using 'top-down' design, and the task of constructing an ADT can be separated from the rest of the application program. The users of the ADT can only manipulate it using the provided operations, through a well-defined and well-controlled interface, since the implementation of the ADT remains hidden from the user, and is therefore inaccessible to him. It can then be changed, if necessary, without affecting the rest of the application.

There are three main methods of describing the *structure* of an ADT, each has its own strengths and weaknesses, described below:

1. Graphics

   The graph is easy to understand, but cannot easily be manipulated using a computer.

2. Natural language

   This method has great power of expression, but is cumbersome to use and does not facilitate manipulation; ambiguity can also result.

3. A formal definition using mathematical notation

   This method has the virtue of precision, economy of expression, avoiding ambiguity and providing ease of manipulation, but readability and interpretation can be a problem.

The three methods are complementary, and so all three have been used to fully describe the structure of the ADT Maintenance_History. The Maintenance History is an **Anthology** of **Books**; in terms of ADTs, the anthology corresponds to an ADT which is constructed from the ADT Linked-List and the ADT Tree. The structure of the Book, i.e. ADT Tree, will be described first, followed by a description of the structure of the Anthology, i.e. the ADT Linked-List. A *complete* description of an ADT must also include the semantics of the ADT, which is obtained by formally specifying the set of operations on it, before it can be regarded as fully-defined; this forms the subject of the next chapter.

7.5.2.1 Description of the structure of the ADT Tree

7.5.2.1.1 Graphical description

A graph is a pictorial representation of an information structure, i.e. it shows the relationships existing between the objects in the database of the ISMSE. A Tree is a special kind of graph, known as a connected, acyclic, directed, graph, containing a finite set of elements called points, nodes or vertices, representing these objects. The term connected means that it is possible to reach any node from any other node; the term acyclic means that distinct nodes are connected by only one path and the term directed means that the graph is an ordered set of nodes and arcs. An arc is a line joining two nodes, the joining implies that a relationship exists between the nodes. Geometrically, the nodes are represented by dots, which may be labelled for identification purposes, and the arcs are represented by lines, joining the nodes. Normally, in a graph, arcs carry arrows to indicate direction, but by convention these arrows are omitted when representing a tree as a directed graph. The Book structure is shown below as a graph in Figure 7.5.

7.5.2.1.2 Natural language description

A book structure is a hierarchy, usually called a tree structure or simply tree; the terms hierarchy and tree can be considered equivalent. A tree is a dynamic data structure, i.e. both the structure *and* the data within the structure can change. The terminology concerning

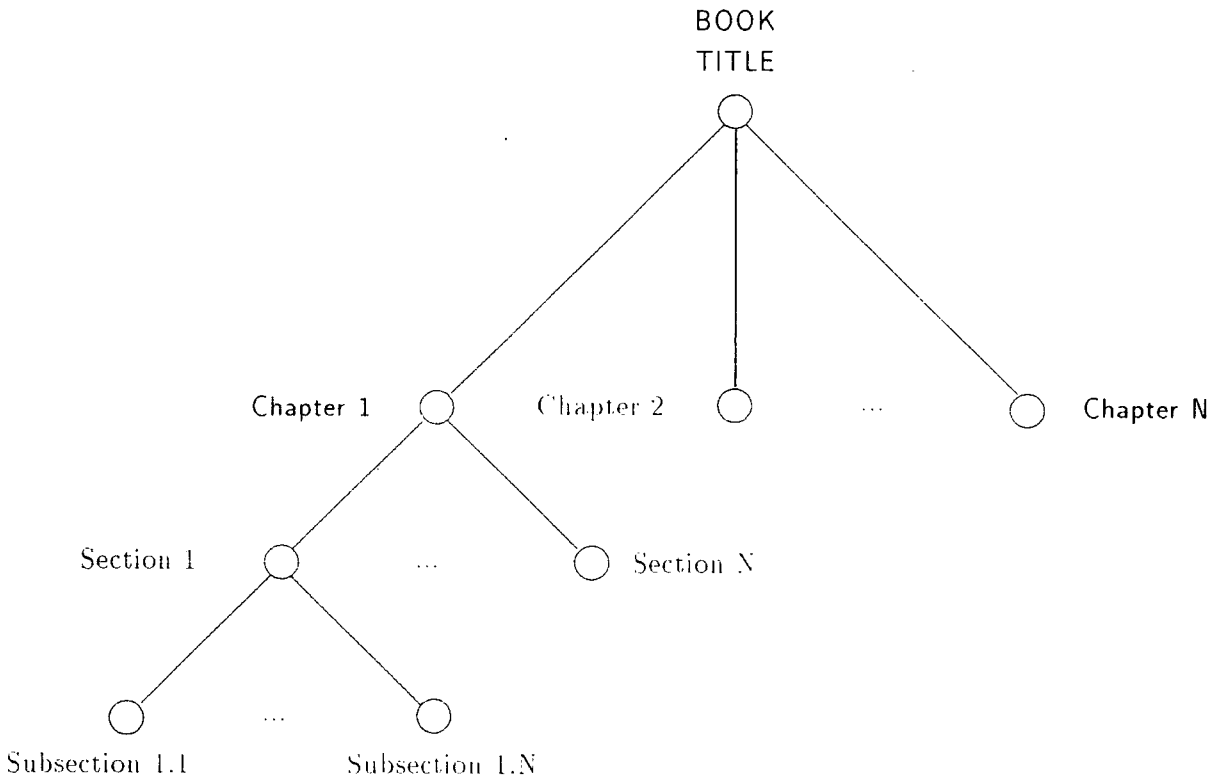


Figure 7.5: The Book Structure as a Directed Graph.

trees is drawn from botany, e.g. forest, root, branch, leaf; from genealogy (family trees), e.g. parent, child, sibling; and from graph theory, e.g. node, arc. Typically, the main elements of a Book are:

1. Title
2. Abstract or Preface
3. Table of Contents
4. Chapters
5. Appendix
6. Index

The natural language description of a Book can be rendered more succinct using a notation provided by a meta-language, such as Backus-Naur, since it acts as a defining mechanism for a language, through a precise description of its syntax, thus reducing the possibility of ambiguity. The production rules for the components of a book are given below in Figure 7.6. This notation shows the decomposition of the fundamental components of the Book into their sub-components, but does not give a clear indication of the hierarchy; however the hierarchical structure of a book is evident from the table of contents shown in Figure 7.7 below, the indentation indicating the hierarchy.

Regarding the Book as the root of the tree, represented by its title, the Book is the parent of n children, each of which is a subtree of the root T, denoted by the label r(T), represented by chapters 1 to N, and each of these subtrees has its own root denoted by the labels r(T1)

$< book >::=< prologuepart >< body >< epiloguepart >$
$< prologuepart >::=< title >< author(s) >< date >< preface >< tableofcontents >$
$< body >::= [< chapters >]_1^N$
$< epiloguepart >::= [< appendix >]_0^N < index >$
$< chapter >::= [< section >]_1^N$
$< section >::= [< subsection >]_1^N$
$< subsection >::= [< textpart >]_0^N [< diagrampart >]_0^N$
$< textpart >::= [< paragraph >]_1^N$
$< paragraph >::= [< subparagraph >]_1^N | [< lines >]_1^N$
$< subparagraph >::= [< lines >]_1^N$

Figure 7.6: Backus-Naur description of Book Structure

```
Table of Contents                        Node Label

   BOOK TITLE ..................   r(T)
      Chapter 1 ................   r(T1)
         Section 1.1 ...........   r(T11)
             Subsection 1.1.1 ..   r(T111)
             ...
             Subsection 1.1.m ..   r(T11m)
        ...
         Section 1.n ...........   r(T1n)
       ...
      Chapter N ................   r(TN)
```

Figure 7.7: Table of Contents showing Hierarchical Nature of the Book Format

to r(TN). Chapter 1 is the parent of n children, each of which is a subtree of the root r(T1), represented by sections 1.1 to 1.n, and each of these subtrees has its own root, denoted by the labels r(T11) to r(T1n). Section 1.1 is the parent of m children, each of which is a subtree of the root r(T11), represented by subsections 1.1.1 to 1.1.m, and each of these subtrees has its own root denoted by the labels r(T111) to r(T11m). Two nodes which are the children of the same parent node are termed siblings; except for the root T, which has no parent, each node has just one parent. Except for the root T, any node may have siblings. For any r(Ti) the next sibling is denoted by r(Ti+1).

7.5.2.1.3 Formal description

From the above description of a book, which possesses a tree structure, the ADT Tree is defined here as a set of zero or more elements called nodes arranged in a hierarchical manner such that:

1. Except when empty there is one node at the highest level, called the root.

2. The remaining nodes are partitioned into zero or more disjoint sets, T1, T2, ... Tn, where each of these sets is itself a tree. The tree is denoted T = (r, T1, T2, T ... Tj) and each tree Ti is an immediate subtree of T. The subtrees of T are T itself and the subtrees of its immediate subtrees.

3. The sets T1, T2, ... Tn are called the subtrees of the root. If the ordering of these trees is significant, then the tree is called an ordered tree, which means that the path from the root of the tree to any node is unique.

4. Every node except the root is joined to just one other node at the next higher level.

5. Information of a predefined type is associated with each node.

6. A predefined relationship exists between nodes on adjacent levels.

7.5.2.2 Description of the structure of the ADT Linked List

7.5.2.2.1 Graphical description

The terminology and conventions concerning graphs was covered when describing the ADT Tree. The ADT Linked-List can be represented as an ordered graph as shown below in Figure 7.8.

7.5.2.2.2 Natural language description

The linked list is a dynamic data structure, i.e. both the information in the structure and the structure itself can change. The data structure contains elements, each element in the list contains a piece of information.

7.5.2.2.3 Formal description

A sequential list is either empty or is a finite ordered tuple $(a_1, a_2, a_3 ... a_n)$ where the $a_i, 1 \leq i \leq n$ are nodes.

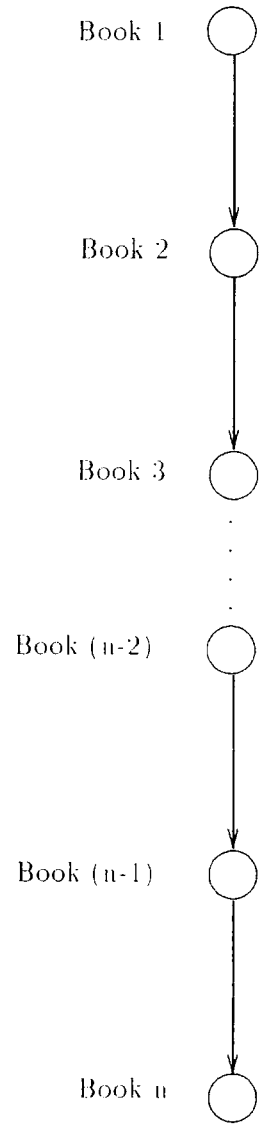


Figure 7.8: The Linked-List structure as a directed graph

7.6 Summary

Some problems encountered in software engineering environments in providing support for automation have been described, as has the role played by the ISMSE in helping to solve these problems. A complementary and complete set of generic tool-types to support the maintenance model has been described and an information structure has been proposed to store information and knowledge concerning the maintenance of software, providing the basis for a documentation paradigm to record a Maintenance History of a software system. The structure of the ADT Maintenance-History has been fully described in terms of its constituent ADTs, the ADT Linked-List and the ADT Tree, using a graphical method, a natural language method and a formal method, its structure is shown in Figure 7.9 below. The ADT Maintenance_History permits a precise description of the database view for the ISMSE, providing the conceptual schema in a three-level database architecture, enabling a bridge to be built between the software tools' views of the information structure, (a prerequisite for accessing and updating the information structure), and the underlying internal schema used for its physical storage.

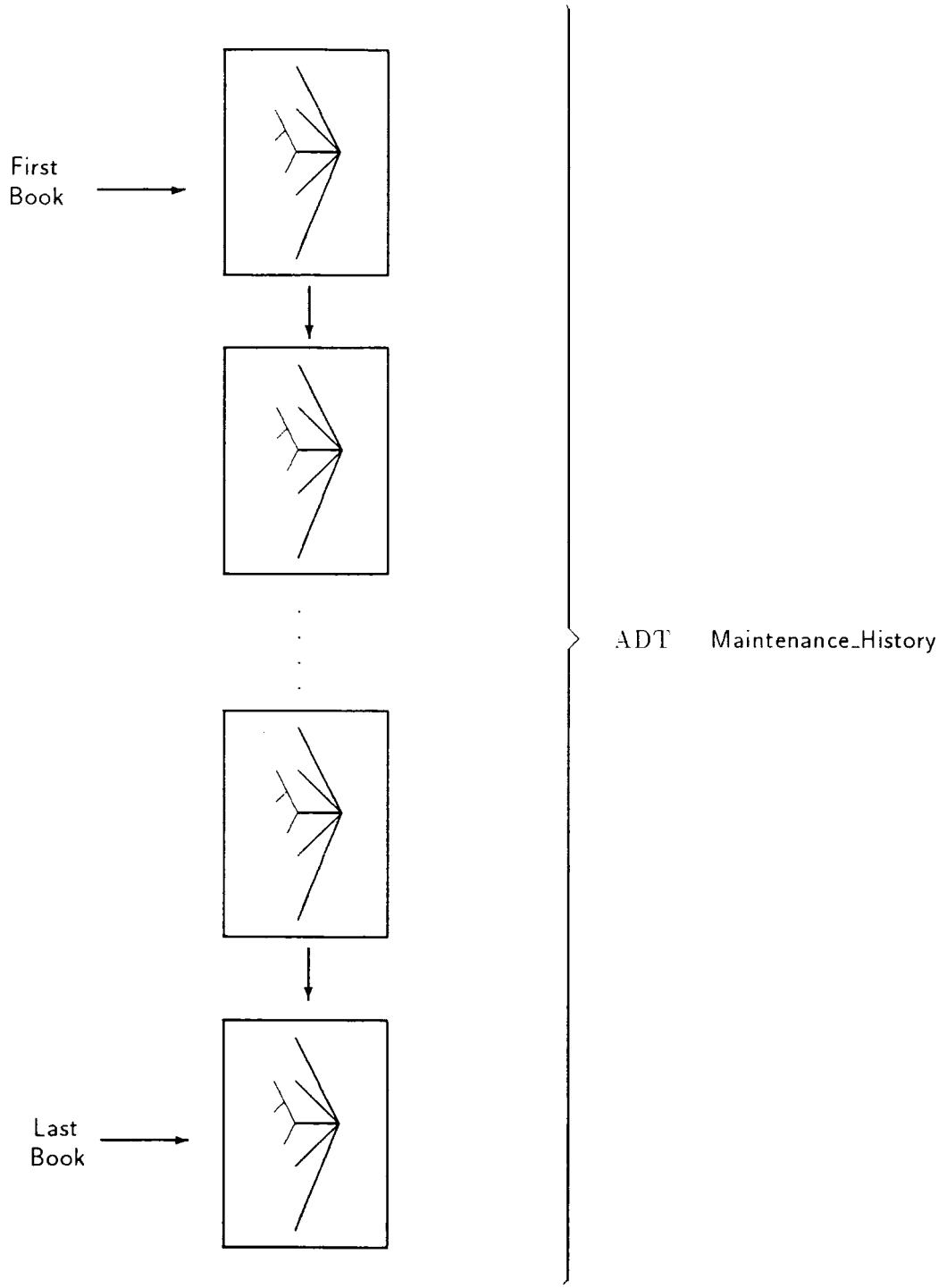


Figure 7.9: The ADT Maintenance_History constructed from the ADT Linked-List and the ADT Tree

Chapter 8

Formal Specification of the ADT Maintenance-History

8.1 Introduction

The problems associated with natural-language specification were discussed in chapter six; briefly, these were concerned with ambiguity, context-sensitivity and possible differences of interpretation. In this chapter a formal approach is used to capture the semantics of the ADT Maintenance-History, through the specification of the operations which define it. The

formal specification of the ADT Maintenance-History is, in essence, a formal definition of each operation on the ADT. Formal specification techniques have a sound theoretical basis and provide a basis for precise reasoning about the behavioural aspects of an ADT. The term 'formal specification' implies that:

1. A specification is expressed using a notation which has, as its foundation, rigorous mathematics.

2. There is a need for both the syntax and the semantics of the specification-language, used to express the formal specification, to be formally defined, so that the *meaning* of a specification can be determined by reference to the specification-language *definition*.

8.2 The benefits provided by the use of formal techniques for specification

These are:

1. **An aid to improving the quality of natural language specifications**

   The formal technique acts as a tool for refining natural language specifications, through the exposure of ambiguities and contradictions. Algebraic specifications are easy to read and understand, facilitating informal and formal verification.

2. **The provision of an intermediate step between requirements and design**

   Specification is concerned with the precise definition of the tasks to be performed by a system. It was the author's experience that the greatest benefit in applying a formal method accrued through the *process* of formalising, making possible the gaining of a deeper understanding of the system being specified, by being abstract and meticulous concerning the desired properties of the system.

3. **The provision of a framework for verifying a system in a systematic manner**

   According to Meyer [84] semantic analysis of formal specifications can be assisted by software tools, making possible machine analysis and manipulation, but this is not possible with informal specifications, i.e. those written using natural language.

   Wing states [128] that some proof-checking tools e.g. Larch, Prover, OBJ, enable algebraic specifications to be treated as re-write rules.

   Guttag [50] has shown that algebraic specifications of data types can play a significant role in *program* verification, permitting factorisation of proofs into distinct manageable stages. He also showed [50] that the use of pure functions and equations as the form of specification permits proofs to be constructed, in large part, as sequences of substitutions, using the equations as rewrite rules.

4. **Support for re-use**

   Meyer states that [84] an essential requirement of a good specification formalism is that it should favour reuse of previously written elements of specifications. The documentation paradigm is specified as an ordered binary tree and therefore supports binary search.

5. **Communication**

   Wing points out [128] that a specification may serve as a contract, which is a valuable piece of documentation, and a means of communication between a client, a specifier and an implementor.

6. **Automatic update of test-suite**

   As mentioned in chapter six, an automatic approach to the identification of obsolete functional tests is only possible where a formal specification language has been used, which permits the analysis of changes in the specification.

8.3 The formal specification of data abstractions

As pointed out by Guttag [51] there are many possible methods of specifying the semantics of an abstract data type: most of these can be classified as operational or definitional. The operational method provides a comparatively easy means of constructing the type but forces the overspecification of the abstraction, whereas the definitional method, although more difficult, avoids this problem.

8.3.1 Operational approach

The operational approach defines the ADT operations in terms of some other known set of operations that are *not* those characterising the ADT. These other operations form a general underlying model, upon which the definition of the ADT is constructed. This model can then be used as the basis for the constructive specification of a range of related ADTs. This underlying model needs to be defined either implicitly, by means of a mathematical technique, such as set theory, or explicitly, using a formal language such as Meta IV [59], Z [119], or algebraic axioms, described below, which is itself a formal language.

8.3.2 Definitional approach

An example of the Definitional approach is an algebraic technique developed by Zilles [133], Guttag [51], and Goguen et al [47]; it has two parts:

1. An interface part which names the allowed operations and specifies the types of their parameters

2. An axioms part which defines the behaviour of these operations.

An object is specified in terms of the relationships between the operations that act on the object, and therefore fits in well with the concept of ADTs. The algebraic approach to specifying ADTs is the definition of its properties as a set of axioms. This approach re-

quires that each operation acting on the ADT should have axioms associated with it, which state what may be asserted after execution of that operation; the assertions being made in terms of what was true before execution. A small set of axioms is sufficient to totally define an ADT; often these axioms enable the deduction of further properties of the object. A formal-specification of an ADT using the algebraic approach involves a specification of the syntax and the semantics of the operations. The algebraic approach is a fundamental one, since it defines the operations of an ADT by *relating their meanings to one another, without reference to any operations other than those characterising the ADT.*

Discussion.

According to Sommerville [118] the model-based approach to formal specification is not yet mature and is not yet widely usable as a software engineering tool, since it does not lend itself to the structuring of specifications at the architectural level. Liskov and Berzins [74], are also of the opinion that the algebraic approach is by far the better-developed of the two specification techniques.

The importance of constructing specifications incrementally is well known, and is a natural way of producing a specification, using the technique of *enrichment*, which allows algebraic specifications to be structured and built out of existing specifications. The new ADT produced by enrichment inherits the operations and axioms defined over the old ADT, so that these apply to the new ADT. Enrichment of algebraic specifications is a powerful abstraction tool for building a formal specification, since it allows higher-level operations on an ADT to be devised, which can then be defined in terms of lower-level operations, whose algebraic

axioms are well-known. In contrast, enrichment cannot easily be used with the constructive approach. For these reasons the algebraic approach has been adopted in this thesis.

8.4 Completeness of Algebraic Specifications

An ADT can be viewed as a store of information *plus* the collection of related operations that can be carried out on it; each operation has three components associated with it; source data, results and the relationship existing between them. The word *related* is important since the meanings of the 'atomic' operations that characterise an ADT are *not* independent of one another. The behaviour of an ADT can only be seen by observing the results of the manipulations on it, i.e. by applying the set of operations which define it, and so the first stage in the formal specification of an ADT is its definition, via the identification of these allowed operations. This is important, since capturing the semantics of the data type helps to ensure that its formal specification is *complete*. The complete capture of the semantics of an ADT can only be achieved if the axioms define operations which allow the construction of all possible instances of the ADT, (an ADT *is* the set of all possible values of the type), and which also define the result of all permissible operations on the ADT. The completeness of the specification draws upon the work of Guttag [50], and Fairley [37]. The complete set of operations defining an ADT is ADT-dependent, but if the total set of operations possible on the ADT Maintenance-History is represented by $p1, \ldots, pn$, and the minimum subset necessary to capture its semantics as $q1, \ldots, qr, r \leq n$, then this subset r can be partitioned into three subsets:

1. Constructor set (x)

2. Behaviour set (y)

3. Modifier set (z)

A sufficiently complete set of axioms necessary to capture the semantics of the ADT is compiled by providing axioms for each member of these three subsets, of the form:

$Behaviour(Constructor()) = ?$

$Modifier(Constructor()) = ?$

From this it can be seen that the total number of axioms which define the ADT is:

$(Card(z) * Card(x)) + (Card(y) * Card(x))$

The definition of axioms setting out the behaviour of ADT operations begins by identifying these 'Constructor', 'Modifier', and 'Behaviour' operations.

Common to all dynamic data structures are two classes of operation: Transformation, whereby the structure and/or content of the information structure is changed, and Navigation, whereby the structure and/or content of the information structure is *un*changed. These two classes of operation may be partitioned into three sub-classes, each of which contains a set of generic lower-level operations, enumerated below. Each generic operation is labelled as a 'Constructor' (C), 'Modifier' (M), or 'Behaviour' (B).

1. **Transformation**

   (a) **Initialisation (C)**

   Creates a new instance of the ADT.

(b) **Assignment (M)**

Changes the value held in an existing element of the ADT or can be used to copy all or part of the structure into another structure.

(c) **Rearrangement (M)**

Re-orders the items within the ADT. This may be done manually, as a 'Prune and Graft' operation or may be automatic, for example Sorting.

(d) **Deletion (M)**

Reduces the size of the ADT.

(e) **Insertion (C)**

Increases the size of the ADT.

2. **Navigation**

(a) **Accessing (B)**

Identification of the required element by virtue of its *position* in the ADT.

(b) **Searching (B)**

Identification of the required element by virtue of its *contents*, a given field in the element acting as the search key.

(c) **Retrieval (B)**

Obtains information previously stored in an element of the ADT using Search and Copy operations, or provides information about the ADT itself, e.g. where to find information within the ADT.

(d) **Browsing (B)**

Moving backwards and forwards within the ADT - this uses the **Write** operation.

which is concerned with Retrieval, and begins with a Search operation and ends with a **Write** operation to standard output or a printer.

(e) **Comparison (B)**

Compares information content of elements of the ADT.

8.5 Consistency of Algebraic Specifications

If any two axioms are contradictory then an algebraic specification is inconsistent: however the fact that such a specification is written formally, i.e. makes use of rigorous mathematics, means that it can be demonstrated that the axioms are not contradictory. Guttag has shown that a recognisably complete axiomatisation can be viewed as a set of replacement rules, and its consistency demonstrated by proving that the set of replacement rules exhibits the Church-Rosser property [26]. Machine verification is also possible for axiomatic specifications: an interactive system is described by Guttag in [50].

8.6 The operations on the ADT Maintenance-History

Informally, the ADT Maintenance-History is an information-structure designed to store information, and to record the relationships between items of information, as well as providing the means of using and changing the information held in it; this is a summary of the require-

ments for an ADT. Formally, the ADT Maintenance-History is of a certain type; each of its components is an ADT, and also has a type associated with it. The ADT Maintenance-History can be regarded as a **collection** of ADTs, and so the operations can be subdivided into those for the Anthology, and those for each Book, contained within the Anthology. Some of the generic operations may not be meaningful for a particular ADT, and some may only be performed conditionally. The complete set of operations on the ADT Maintenance-History is intended to reflect the maintenance model adopted for an ISMSE, and in addition it is intended, as far as is practicable, to mirror all the manual ways in which it is possible to use a 'hard-copy' counterpart of the ADT.

8.6.1 The operations on the ADT Anthology

1. **Transformation**

   (a) **Initialisation**

   A new empty instance of the ADT Anthology can be created using the **Create** operation.

   (b) **Assignment**

   A software system evolves during its lifetime, giving rise to successive versions. It may be decided during the lifetime of the software that support for early versions of the software is no longer a viable proposition. The earliest version for which support is to be provided is known as the 'baseline' version, and this 'baseline' is moved forward to a later version as successive versions of the software are

produced. The element of the ADT Anthology which is to serve as this 'baseline' version can be designated using the **Assign** operation.

(c) **Rearrangement**

The order of the elements in the Anthology can be changed using the operations below.

i. **Deletion**

An element, i.e. a Book with a given version number can be removed from the Anthology using the **Delete** operation.

ii. **Insertion**

An element, i.e. a Book with a given version number can be inserted at a given position into the Anthology using the **Insert** operation.

Initialisation, Assignment and Rearrangement could be used for updating a table of contents and a master index for the Anthology.

2. **Navigation**

(a) **Accessing**

Evaluation of a particular Book version number corresponding to its position in the Anthology, using the **Evaluate** operation.

(b) **Searching**

Find the position of a given element in the Anthology using its version number, which acts as the search key, using the **Position** operation.

(c) **Retrieval**

Retrieve the latest element in the Anthology, using the **Latest** operation. The

latest book in the Anthology is designated as the one which corresponds to the latest version of the software. If it is required to insert a Book with a given version number into the Anthology then it must be ascertained whether that version number already exists: using the operation **Isin_Anthology**.

(d) **Browsing**

Given the position of an element in the Anthology, find the next or previous element in the Anthology, using the **Next** and **Previous** operations, and Display the resulting version number. This makes possible moving from Book to Book, i.e. browsing, with the ability to return to the index or table of contents at any time from any Book.

ANTHOLOGY ($Elem : [Undefined \rightarrow Elem]$)

sort: Ordered-List
imports: integer, boolean

Description of sort and operations

This specification defines the ADT Anthology, which is an enrichment of the sort Ordered_list, with its members arranged in ascending order. It inherits the operations of the ADT Ordered_list, but the operations to construct the Ordered_list and to add a member to the list are hidden, i.e. would not be accessible in any implementation, to ensure that the ordering of the Anthology cannot be compromised.
The constructor operation produces an Anthology containing one item. The ordering of the list is maintained using the insert operation.

SETS

A = {a: a is an Anthology}
N = {n: n is a Book title}
Z = {z: z $\geq$ 1 }

Syntax:

$Con\_Anthology : \rightarrow A$

Semantics:

Con_Anthology(n,z,a) = append(make(n,z),a)

Figure 8.1: Algebraic specification of ADT Anthology

ANTHOLOGY ($Elem : [Undefined \rightarrow Elem]$)

SETS

A = {a: a is an Anthology}
N = {n: n is a Book title}
B = {*true*, *false*}
Z = {z: z $\geq$ 1 }
M = {m: *Book not present*, *Out of range*, *Book already present*, *Next Book does not exist*, *Previous Book does not exist*}
F = {f: f is an output file}

Syntax:

$Create\_Anthology : \rightarrow A$
$Isin\_Anthology : Z \times A \rightarrow B$
$Delete : N \times A \rightarrow A \cup M$
$Insert : Z \times N \times A \rightarrow A \cup M$
$Position : N \times A \rightarrow Z \cup M$
$Evaluate : Z \times A \rightarrow N \cup M$
$Write : A \rightarrow F$
$Assign : A \rightarrow A$
$Next : Z \times A \rightarrow N \cup M$
$Previous : Z \times A \rightarrow N \cup M$
$Latest : A \rightarrow N \cup M$
$Earliest : A \rightarrow N \cup M$

Figure 8.2: Algebraic specification of ADT Anthology (contd.)

ANTHOLOGY ($Elem : [Undefined \rightarrow Elem]$)

Semantics:

$\forall a \in A, \forall b, n \in N, \forall k, z \in Z$:

Isin_Anthology(b,Create_Anthology) = false ...(**A1**)

```
Isin_Anthology(b,Con_Anthology(n,z,a)) = if b = n
                                         then
                                             true
                                         else
                                             if b < n
                                             then
                                                 false
                                             else
                                                 Isin_Anthology(b,a) ...(A2)
```

Delete(b,Create_Anthology) = 'Book not present' ...(**A3**)

```
Delete(b,Con_Anthology(n,z,a)) = if b = n
                                 then
                                     a
                                 else
                                     if b < n
                                     then
                                         'Book not present'
                                     else
                                         Con_Anthology(n,z,Delete(b,a)) ...(A4)
```

Insert(b,Create_Anthology) = Con_Anthology(b,Create_Anthology) ...(**A5**)

Figure 8.3: Algebraic specification of ADT Anthology (contd.)

ANTHOLOGY ($Elem : [Undefined \rightarrow Elem]$)

Semantics:

$\forall a \in A, \forall b, n \in N, \forall k, z \in Z$:

```
Insert(b,Con_Anthology(n,z,a)) = if b = n
                                 then
                                     'Book exists'
                                 else
                                     Con_Anthology(n,z,Insert(b,a)) ...(A6)

Position(b,Create_Anthology) = 'Book not present' ...(A7)

Position(b,Con_Anthology(n,z,a) = if b = n
                                  then
                                      z
                                  else
                                      Position(b,a) ...(A8)

Evaluate(k,Create_Anthology) = 'Book not present' ...(A9)

Evaluate(k,Con_Anthology(n,z,a)) = if k = z
                                   then
                                       n
                                   else
                                       if k > 0
                                       then
                                           Evaluate(k,a) ...(A10)
```

Figure 8.4: Algebraic specification of ADT Anthology (contd.)

ANTHOLOGY ($Elem : [Undefined \rightarrow Elem]$)

Semantics:

$\forall a \in A, \forall b, n \in N, \forall z \in Z$:

Write(Create_Anthology) = Create_Anthology ...(**A11**)

Write(Con_Anthology(n,z,a)) = Write(n); Write(a) ...(**A12**)

Assign(b,Create_Anthology) = Create_Anthology ...(**A13**)

Assign(b,Con_Anthology(n,z,a)) = (Con_Anthology(n,z,a))...(**A14**)

Next(Create_Anthology) = 'Book not present' ...(**A15**)

Next(Con_Anthology(n,z,a)) =
if not(a = Create_Anthology) then
    Evaluate(succ(Position(n,Con_Anthology(n,z,a))),Con_Anthology(n,z,a))
    ...(**A16**)

Previous(Create_Anthology) = 'Book not present' ...(**A17**)

Previous(Con_Anthology(n,z,a)) =
if not(z = 1)
then
    Evaluate(pred(Position(n,Con_Anthology(n,z,a))),Con_Anthology(n,z,a))
    ...(**A18**)

Earliest(Create_Anthology) = 'Book not present' ...(**A19**)

Earliest(Con_Anthology(n,z,a)) = n ...(**A20**)

Latest(Create_Anthology) = 'Book not present' ...(**A21**)

Latest(Con_Anthology(n,z,a)) = if (a = Create_Anthology)
    then
        n
    else
        Latest(a) ...(**A22**)

Figure 8.5: Algebraic specification of ADT Anthology (contd.)

8.6.2 Natural language description of axioms for ADT Anthology.

(A1) simply states that the empty Anthology contains no Books.

(A2) makes use of the fact that the Books in the Anthology are strictly ordered according to version code, which is an alphanumeric key, used as the title of the Book, and also the key for all search operations: the first Book in the list having the smallest version number. The operation Con_Anthology acts as a deconstructor, so that the Anthology can be regarded as being comprised of the first Book, followed by the remainder of the Anthology. If the remainder of the Anthology is empty then axiom (A1) can be applied. If the Anthology contains more than one Book then the list is searched recursively, termination occurring either by a successful search or by the key for the search being out of range, or by there being no more Books with which to compare keys, whereupon axiom (A1) applies.

(A3) states that a Book cannot be deleted from an empty Anthology.

(A4) states that a Book can only be deleted if its key is within the range of keys present in the Anthology. If the Book is not the first in the Anthology, then the remainder of the Anthology must be searched recursively, termination occurring either by a successful search, or by the key for the search being out of range, or by there being no more Books with which to compare keys, when axiom (A3) applies.

(A5) states that insertion of a Book into an empty Anthology is achieved by constructing an Anthology containing just one Book.

(A6) states that if the Anthology is not empty then the insertion of the book into the Anthology is only possible if the title does not already exist within the Anthology, otherwise

an error message is output. The correct position in the Anthology is found using a recursive search of the keys of the other Books in the Anthology. If the position for insertion is at the end of the Anthology then (A5) applies.

(A7) states that Position in an empty Anthology is undefined, and so an error message is output to this effect.

(A8) states that if the key of the Book sought is not equal to the key of the current book in the Anthology, the search key is compared recursively with those of the remaining Books in the Anthology. Termination of the search occurs, either because a match is found, or all Books have been compared without a match, whereupon (A7) applies.

(A9) states that version code is undefined for an empty Anthology and so a message is output to this effect.

(A10) states that the result of attempting to recover a version code at a given position in the Anthology is undefined if the position is not a member of the set of natural numbers greater than zero, or is greater than the cardinality of the set of Books in the Anthology

(A11) states that writing an empty Anthology produces an empty file. **(A12)** states that writing an Anthology containing Books produces a file in which the key of the latest member is written first, followed by those of the remainder, in order of their version codes.

(A13) states that attempting to assign a version code to an empty Anthology has no effect.

(A14) states that assigning a version code to a Book in the Anthology does not affect the structure of the Anthology.

(A15) states that searching an empty Anthology for a version code of the next element in the Anthology is undefined and a message is output to this effect.

(A16) states that the version code returned by this operation is that of the element whose

position is the one following the current position, except when the current position is the last position in the Anthology.

(A17) states that searching an empty Anthology for a version code of the previous element is undefined and a message is output to this effect.

(A18) states that the position passed as a parameter to the operation is the predecessor to the current position, except when the current position is the first position in the Anthology.

(A19) states that an attempt to find the earliest Book in an empty Anthology is undefined and a message is output to this effect.

(A20) states that the earliest Book in an Anthology which is not empty is simply that produced by the deconstructor operation.

(A21) states that an attempt to find the latest Book in an empty Anthology is undefined and a message is output to this effect.

(A22) states that the latest Book in an Anthology which contains *more* than one Book is found by applying the operation recursively to the Anthology.

8.6.3 The Operations on the ADT Book

1. **Transformation**

   (a) **Initialisation**

   This operation enables the construction of the book: this is achieved via the **Create** operation. From the change request the Anthology must be searched to see whether the change request is 'valid', i.e. the problem may be on file, and

maintenance may be ongoing or a solution may have been found to the problem which is contained in another version of the software.

(b) **Assignment**

An item of information is placed into an element of the book, using the **Assign** operation.

(c) **Rearrangement**

i. **Deletion**

A piece of information can be removed from the book using the **Delete** operation.

ii. **Insertion**

The **Insert** operation can be used to insert the components of the book.

iii. **Editing**

The Book format can be edited using the **Graft** and **Prune** operations to reflect copying, moving and deleting pieces of information to simulate 'cut and paste' operations - this enables partitioning or repartitioning a Book into chapters, and sections.

2. **Navigation**

(a) **Accessing**

A particular component of the Book can be accessed by virtue of its unique alphanumeric key, using the **Evaluate** operation.

(b) **Searching**

A pattern-matching operation is used for content-search, i.e. searching text; a

component of the book is the subject of the search, the string to be searched for is the parameter supplied to the **Cross_reference** operation. Traceability between phases of the software life-cycle can be verified, by finding references to a variable in e.g. requirements, specification, design and source code, using the **Evaluate** operation.

(c) **Retrieval**

i. This operation could be used to map inputs and outputs from source code modules to the data dictionary to aid in documentation.

ii. Information about a particular aspect of the ADT Book, e.g. chapter headings or table of contents, can be obtained using the **Write** operation.

iii. Change requests can be listed, displayed, or printed, using the **Write** operation.

iv. Re-use of modules and associated documentation can be accomplished using the **Copy** operation.

v. The **Evaluate** operation provides a means of recording which parts of the Anthology have been visited. If a number of books have been perused during the course of a maintenance assignment, then recording these books and the components of each Book which have been of interest enables the maintainer to reuse this 'virtual' Book, when continuing the maintenance assignment, by using the **Trail** operation.

(d) **Browsing**

The following modes of access must be supported.

i. The ability to get to the index or table of contents at any time from anywhere is made possible using the **Home** operation. Then any part of any Book can be accessed from any other part of any other Book. The **Bookmark** operation makes it possible to keep the current place in the Book.

(e) **Comparison**

i. Comparing the module structure chart of *versions* of source code, using the **Equal** operation to reveal the differences in their structures.

ii. A documentation-module structure can be constructed using the **Copy** operation to mirror the call-graph structure of the source code. Then a comparison of the keys associated with each node in each tree using the **Equal** operation provides a means of ensuring that the documentation structure has been modified to reflect any change to the source code's call-graph structure.

The Book structure is that of an m-ary tree, and because the operations on such a tree are dependent on the order of the tree, they are complicated and non-standardised. For this reason the m-ary tree is transformed into a Knuth ordered binary tree, as shown in Figure 8.6 below, prior to specifying the axioms for the operations on the Book. The resulting Knuth ordered binary tree is logically equivalent to the original m-ary tree, there being no loss of information during the transformation. The relationships between the nodes in the Knuth ordered binary tree are shown in Figure 8.7 below. The advantages of specifying the axioms on the Knuth ordered binary tree are described below.

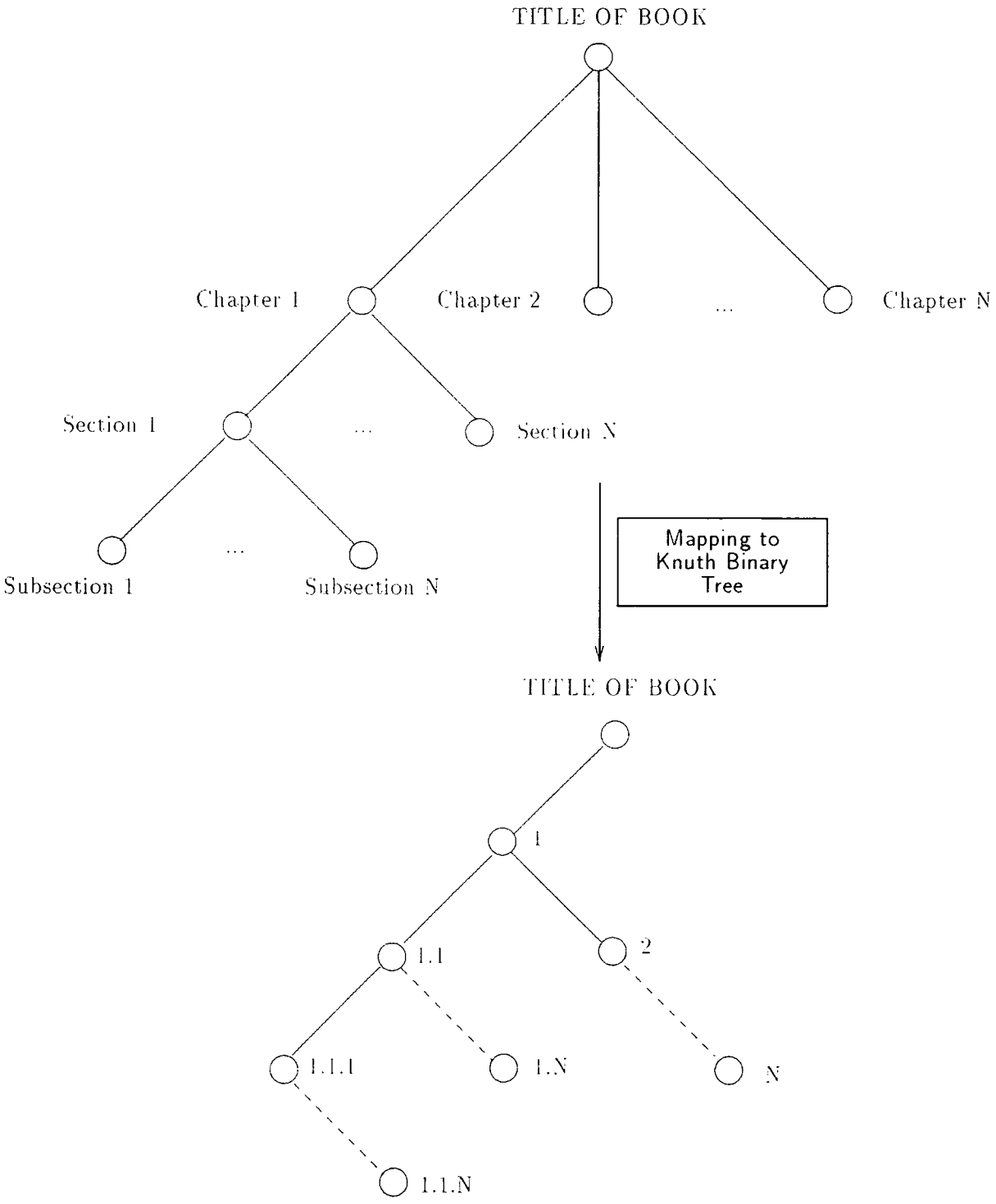


Figure 8.6: Conversion of m-ary tree to Knuth ordered binary tree

1. The *structure* of a binary tree is completely recursive, allowing a recursive definition of the axioms specifying the operations; the transformation from m-ary to binary is, therefore, a powerful abstraction tool.

2. Searching

   Searching in a binary tree is rapid and since the key of the node is related to the contents of the node, then the search can be content-controlled.

3. Traversals

   The traversal of a binary tree produces a list, whose structure depends on the structure of the tree, and the type of traversal employed, providing information concerning the structure of the tree, e.g. table of contents.

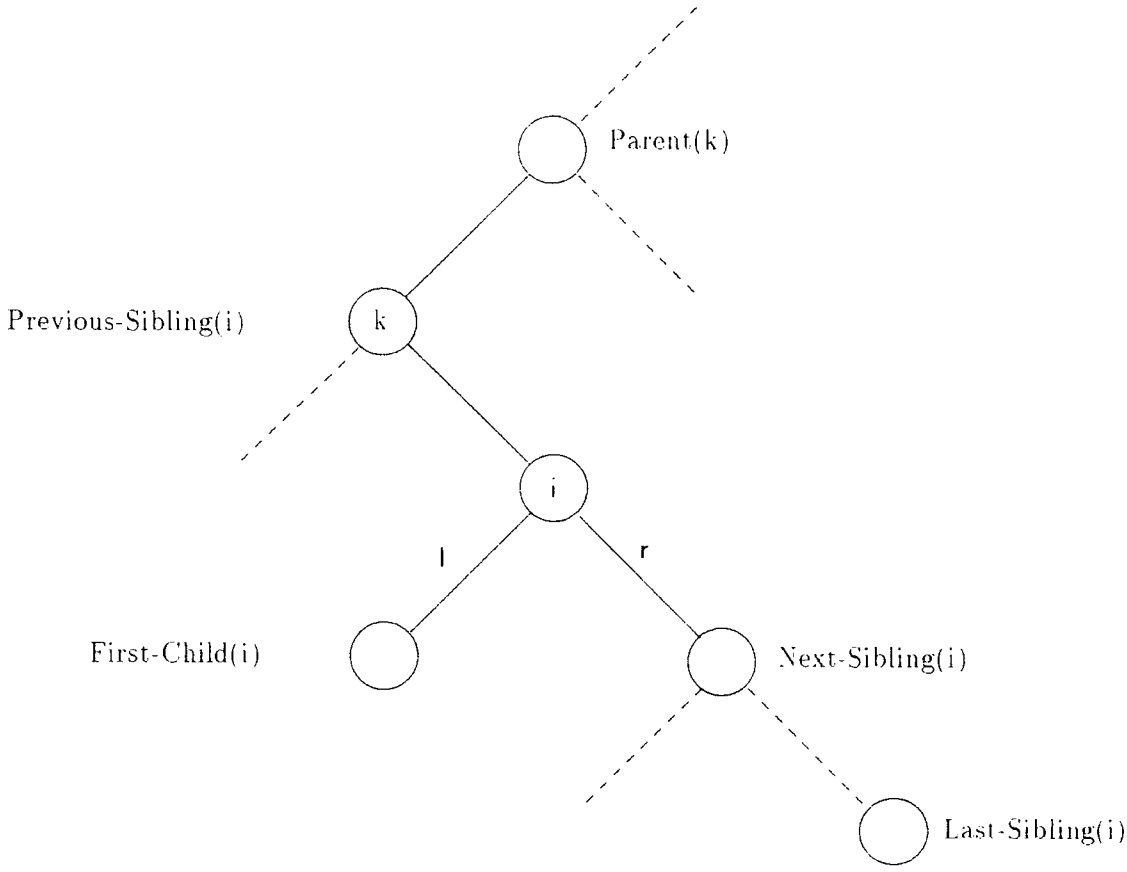


Figure 8.7: Relationships between nodes in the Knuth ordered binary tree

BOOK ($Elem : [Undefined \rightarrow Elem]$)

sort: Knuth Binary Search Tree
imports: integer, boolean

Description of sort and operations

This specification defines the ADT Book, which is, predominantly, an enrichment of the sort Binary Search Tree. It inherits some of the operations of the ADT Binary Search Tree, and also of the ADT Queue. The operations for the Queue are considered first.

An operation **Get_Next** is required which returns the item at the front of the queue, and then deletes this item. The output from this function is specified as a tuple. Displaying the contents of the Queue makes use of the **Write** operation. The last item in the queue is returned using the **Last** operation.

SETS

I = set of items
Q = set of queues
F = set of files

Syntax:

$Get\_Next(Queue) : Q \rightarrow I \times Q$
$Write(Queue) : Q \rightarrow F$
$Last(Queue) : Q \rightarrow I$

Figure 8.8: Algebraic specification of ADT Book

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics:

$\forall\, i \in I. \forall\, q \in Q$:

Get_Next(Create_Queue) = ('The queue is empty',Create_Queue)

Get_Next(Add_to_Queue(i,q)) = (Front(q),Delete_From_Queue(q))

Write(Create_Queue) = 'Empty Queue'

Write(Add_To_Queue(i,q)) = Write(Get_Next(Add_to_Queue(i,q)))

Last(Create_Queue) = 'Empty Queue'

Last(Add_To_Queue(i,q)) = i

The operations relating to the enrichment of the Binary Search Tree are now considered. The operation to construct the Binary Search Tree and to add an item to the Binary Search Tree are hidden, i.e. would not be accessible in any implementation, to ensure that the ordering of the Binary Search Tree cannot be compromised. The ordering of the Binary Search Tree is maintained using the **Insert** operation.

SETS

B = {*true, false*}
T = {t: t is a Knuth binary search tree}
M = {m: *Empty Book, Not equal, ROOT not present*}
N = {n: n is a title}
Z = {z: z is a node key }
I = {i: i is an item }
F = {f: f is an output file}
Q = {q: q is a queue}
K = {k: k is a search key}
E = {e: e is a search string}
$\aleph$ = {n: n>0 }

Figure 8.9: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Syntax:

$Create\_Book : \rightarrow T$
$Make\_Book : T \times I \times T \rightarrow T$
$Assign : Z \times N \times T \rightarrow T$
$Table\_of\_Contents : T \rightarrow F$
$Path : T \rightarrow F$
$Insert : T \times N \times T \rightarrow T \cup M$
$Graft : T \times T \rightarrow T$
$Prune : T \rightarrow T \times T$
$Evaluate : Z \times T \rightarrow (N \cup M) \times Q_e \times F_e$
$Isin\_Book : K \times T \rightarrow B$
$Cross\_reference : T \rightarrow N_1 ... N_i$
$Abstract : T \rightarrow N_1 ... N_i$
$Copy : T \rightarrow T \times T$
$Trail : N \times N ... \times N \rightarrow N_1 ... N_i$
$Home : T \rightarrow N$
$Bookmark : T \rightarrow N$
$Parent : N \times T \rightarrow N$
$Next\_Sibling : N \times T \rightarrow N$
$Previous\_Sibling : N \times T \rightarrow N$
$Last\_Sibling : N \times T \rightarrow N$
$First\_Child : N \times T \rightarrow N$
$Equal : T \times T \rightarrow B$

Figure 8.10: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i,k \in I,\ \forall\, c,n \in \mathbb{N},\ \forall\, e \in E,\ \forall\, l,r \in T,\ \forall\, P \in Q$:

Assign(Create_Book) = 'Empty Book' ...(**B1**)

Assign(k,Make_Book(l,i,r)) = Make_Book(l,i,r) ...(**B2**)

Table_of_Contents(Create_Book) = 'Empty Book' ...(**B3**)

Table_of_Contents(Make_Book(l,i,r)) = Write(PreOrder(Make_Book(l,i,r))) ...(**B4**)

Path(k,Create_Book) = 'Empty Book' ...(**B5**)

```
Path(k,Make_Book(l,i,r)) = if Isin_Book(k,Make_Book(l,i,r))
                           then
                               if k ∈ ℕ
                               then
                                   Write(ROOT)
                               else
                                   Write(k,Path(Parent(k),Make_Book(l,i,r))) ...(B6)
```

Figure 8.11: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i, k \in I,\ \forall\, c, n \in \aleph,\ \forall\, e \in E,\ \forall\, l, r \in T,\ \forall\, P \in Q$:

```
Insert(k,Create_Book) = Make_Book(Create_Book,k,Create_Book) ...(B7)

Insert(k,Make_Book(l,i,r) = if k = i
                           then
                               Make_Book(l,i,r))
                           else
                               if Isin_Book(ROOT,Make_Book(l,i,r))
                               then
                                   if Isin_Book(Parent(k),Make_Book(l,i,r))
                                     or Isin_Book(Previous_Sibling(k),Make_Book(l,i,r))
                                   then
                                       if k < i
                                       then
                                           Make_Book(Insert(k,l),i,r)
                                       else
                                           if k > i
                                           then
                                               Make_Book(l,i,Insert(k,r))
                                   else
                                       Make_Book(l,i,r)
                               else
                                   'ROOT not present' ...(B8)
```

Figure 8.12: Algebraic specification of ADT Book (contd.)

BOOK $(Elem : [Undefined \rightarrow Elem])$

Semantics

$\forall\, i, k \in I,\ \forall\, c, n \in \aleph,\ \forall\, e \in E,\ \forall\, l, r \in T,\ \forall\, P \in Q$:

Graft(T,Create_Book) = T ...(**B9**)

Graft(PreOrder(T),Make_Book(l,i,r)) =
If not(Is_Empty(PreOrder(T))
then
    Insert(Front(PreOrder(T)),Make_Book(l,i,r),
    Graft(Delete_from_Queue(PreOrder(T),Make_Book(l,i,r)))) ...(**B10**)

Prune(Create_Book) = 'Empty Book' ...(**B11**)

Prune(k,Make_Book(l,i,r)) = if Is_in(k,Make_Book(l,i,r))
                            then
                                if i = Data(right(Make_Book(l,Previous_Sibling(k),r)))
                                then
                                    right(Make_Book(l,Previous_Sibling(k),r))
                                        = Create_Book
                                else
                                    if i = Data(Left(Make_Book(l,Parent(k),r))
                                    then
                                        Left(Make_Book(l,Parent(k),r)) = Create_Book
                                                                    ...(**B12**)

Figure 8.13: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i,k \in I,\ \forall\, c,n \in \aleph,\ \forall\, e \in E,\ \forall\, l,r \in T,\ \forall\, P \in Q$:

Evaluate(k.Create_Book) = 'Empty Book' ...(**B13**)

Evaluate(k.Make_Book(l,i,r)) = if k = i
then
Write(Add_to_Queue(Data(Make_Book(l,i,r).Q_e))):
else
if k < i
then
Evaluate(k.l)
else
Evaluate(k.r) ...(**B14**)

Isin_Book(k.Create_Book) = 'Empty Book' ...(**B15**)

Isin_Book(k,Make_Book(l,i,r) = if k = i
then
true
else
if k < i
then
Isin_Book(k,(l))
else
Isin_Book(k,(r)) ...(**B16**)

Cross_reference(e.Create_Book) = Create_queue ...(**B17**)

Figure 8.14: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i, k \in I,\ \forall\, c, n \in \aleph,\ \forall\, e \in E,\ \forall\, l, r \in T,\ \forall\, P \in Q$:

Cross_reference(e,Make_Book(l,i,r)) =
if e $\in$ Data(Make_Book(l,i,r)) then
Appendqueue(Appendqueue(Addtoqueue(i,Create_queue),Cross_reference(l)),
Cross_reference(r)) ...(**B18**)

Abstract(k,Create_Book) = 'Empty Book' ...(**B19**)

Abstract(k,Make_Book(l,i,r)) = if (i $\leq$ k)
then
(Write(i),Abstract(k,l))
else
Abstract(k,l) ...(**B20**)

Copy(Create_Book) = (Create_Book,Create_Book) ...(**B21**)

Copy(Make_Book(l,i,r)) = (Make_Book(l,i,r),
Graft(PreOrder(Make_Book(l,i,r),Create_Book))) ...(**B22**)

Trail(Create_Book) = 'Empty Book' ...(**B23**)

Trail(Make_Book(l,i,r)) = If not IsEmpty(Q_e)
then
Write(Q_e) ...(**B24**)

Figure 8.15: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i,k \in I,\ \forall\, c,n \in \aleph,\ \forall\, e \in E,\ \forall\, l,r \in T,\ \forall\, P \in Q$:

Home(Create_Book) = 'Empty Book' ...(**B25**)

Home(Make_Book(l,i,r) = i ...(**B26**)

Bookmark(Create_Book) = 'Empty Book' ...(**B27**)

Bookmark(Make_Book(l,i,r) = Last(Q_e) ...(**B28**)

Parent(k,Create_Book) = 'Empty Book' ...(**B29**)

```
Parent(k,Make_Book(l,i,r)) = if (Isin_Book(k,Make_Book(l,i,r)))
                             then
                                 if k = c.l
                                 then
                                     c
                                 else
                                     if k = l
                                     then
                                         ROOT ...(B30)
```

Next_Sibling(k,n,Create_Book) = Create_Book ...(**B31**)

```
Next_Sibling(k,n,Make_Book(l,i,r)) = if Isin_Book(k,n,Make_Book(l,i,r))
                                     then
                                         if not(Right(Make_Book(l,i,r))
                                            = Create_Book)
                                         then
                                             k.succ(n) ...(B32)
```

Figure 8.16: Algebraic specification of ADT Book (contd.)

BOOK ($Elem : [Undefined \rightarrow Elem]$)

Semantics

$\forall\, i, k \in I. \forall\, c, n \in \aleph.\ \forall\, e \in E.\ \forall\, l, r \in T,\ \forall\, P \in Q$:

Previous_Sibling(k,n,Create_Book) = Create_Book ...**(B33)**

Previous_Sibling(Next_Sibling(k,Make_Book(l,i,r)) = k ...**(B34)**

Last_Sibling(k,Create_Book) = Create_Book...**(B35)**

```
Last_Sibling(k,Make_Book(l,i,r)) =
if Isin_Book(k,Make_Book(l,i,r))
then
    if Right(Make_Book(l,i,r)
        = Create_Book)
    then
        i
    else
        Last_Sibling(k,r)...(B36)
```

First_Child(k,Create_Book) = Create_Book ...**(B37)**

```
First_Child(k,Make_Book(l,i,r)) = if Isin_Book(k,Make_Book(l,i,r))
                                  then
                                      if k = ROOT
                                      then
                                          l
                                      else
                                          k.1 ...(B38)
```

Equal(Create_Book) = 'Empty Book' ...**(B39)**

```
Equal(Make_Book(T,i,T/) = if not(Inorder(Left(Make_Book(T,i,T/)))
                                  = Inorder(Right(Make_Book(T,i,T/))))
                          then
                              'Not equal' ...(B40)
```

Figure 8.17: Algebraic specification of ADT Book (contd.)

8.6.4 Natural language description of the axioms specifying the operations on the ADT Book

B1 states that an attempt to assign a value to a non-existent node fails and a message is output.

B2 states that assigning a value to the heading of a node has no effect on the *structure* of the binary tree.

B3 states that an attempt to produce a Table_of_Contents from an empty Book fails and a message is output.

B4 states that a Table_of_Contents is a list of nodes in depth-first order, which is produced by a Pre-order traversal of the binary tree. The Write operation sends the list of nodes to a file.

B5 states that an attempt to produce a list of nodes in a file from an empty Book fails and a message is output.

B6 states that a Path is a list of nodes in an output file, the first node in the file is the target node and the last node is the ROOT node.

B7 states that inserting a node into an empty subtree produces a subtree rooted at that node.

B8 States that inserting a node into a non-empty binary tree is only possible if ROOT is present in the binary tree, and either the Parent or the Previous_Sibling of the node is also present in the binary tree.

B9 states that grafting the nodes of a binary tree into an empty binary tree, produces a new binary tree, whose root is the root node of the inserted binary tree.

B10 states that grafting the nodes of a binary tree into a non-empty binary tree, succeeds if the node to be grafted does not already exist in the tree, each node being inserted as a leaf node.

B11 states that an attempt to prune a binary tree which is empty fails, and a message is output.

B12 states that the result of pruning a node is that all the nodes which are attached to that node are also deleted..

B13 states that an attempt to display the contents of a node which does not exist fails and an message is output.

B14 states that displaying the contents of a node is accompanied by the addition of the node key to a queue, as well as its contents being sent to a file.

B15 states that the result of searching for a given node in an empty tree is false.

B16 states that the result of searching for a given node in an non-empty binary tree is true if the node is found and false if it is not found.

B17 states that an attempt to display a cross-reference for a search string in an empty binary tree fails and a message is output.

B18 states that an attempt to display a cross-reference for a search string succeeds if the string is found in a node of the binary tree; the contents of the node are then written to a file.

B19 states that an attempt to produce an abstract from an empty binary tree fails and a message is output.

B20 states that those nodes which comprise an abstract of a given node include that node and all other nodes which are subordinate to the given node.

B21 states that copying an empty tree produces another empty tree.

B22 states that copying a non-empty binary tree is the same as grafting this tree into an empty binary tree.

B23 states that an attempt to output a Trail for an empty Book fails and a message is output.

B24 states that if the binary tree is not empty the contents of the queue are written to a file.

B25 states that an attempt to label the root node of an empty tree fails and a message is output.

B26 states that the label of the root node of a non-empty binary tree is the data item which is used in the operation to construct a binary tree.

B27 states that an attempt to output the key of the last node to be accessed in an empty Book fails and a message is output.

B28 states that the last node to be accessed is the same as the last node inserted into the queue by the Evaluate operation.

B29 states that the Parent of a node is undefined for an empty Book and a message is output.

B30 states that the Parent of a node depends on the type of the node key.

B31 states that the root of the right subtree of a given node is undefined for an empty tree and a message is output.

B32 states that the key of the next sibling of a given node is the root of the right subtree of the binary tree, unless this is an empty tree.

B33 states that the previous sibling of a given node is undefined for an empty binary tree

and a message is output.

B34 states that the operation to return the previous sibling of a node is the inverse of the operation to return the next sibling of a node.

B35 states that last sibling of a node is undefined for an empty tree and a message is output.

B36 states that the last sibling of a node is that node whose right subtree is empty.

B37 states that the first child of a node is undefined for an empty tree and a message is output.

B38 states that the first child of a node is dependent on the type of the node.

B39 states that an attempt to compare the subtrees of an empty Book fails and a message is output.

B40 states that the comparison of two binary trees can be effected by combining them into one binary tree and comparing the left and right subtrees by traversal of the subtrees.

8.7 Summary

The reasons for a *formal* specification of an ADT have been given, and alternative approaches to formal specification have been outlined. A generic set of operations for the ADT Maintenance_History which underlies the documentation paradigm for the ISMSE has been described, and instantiations of these operations have been specified for this ADT, thus capturing its semantics.

Chapter 9

Implementation of the Documentation Paradigm

9.1 Choice of language for the implementation

The obvious candidates for a language to animate the formal specification of the ADT Maintenance History were the functional programming languages, such as LISP, Miranda, Prolog, OBJ, Larch.

There are three main criteria to be borne in mind when choosing a language for an implementation of a specification:

1. The size of the data structure
2. The efficiency of the operations on that structure
3. The complexity of the programs

The first two criteria can be ignored since the implementation was effected using a small problem. The axioms defining the semantics of the ADT were, for the most part, simple and a language was required which could reflect this simplicity. This required, in turn, that the language itself should be axiomatic.

As pointed out by Clocksin, [28] Prolog is based on the idea of a theorem prover, and the basis of a theorem is a set of axioms. Prolog is therefore complementary in this respect to the axiomatic specification used to define the semantics of the documentation paradigm, because of the declarative nature of the language. Prolog also provides a rapid prototyping capability.

Prolog programs are like hypotheses about a known 'world', and questions asked are like theorems which need to be proved or disproved. Prolog is based on first-order predicate calculus, and the Horn clause which expresses a fact or rule in the Prolog database is a statement of independent truth, i.e. an *axiom*, which is independent of what other facts and rules there may be in the database.

9.2 Prototyping the ADT Maintenance_History

Prolog is a declarative language, but it is also possible to write Prolog in a procedural way, as for a block-structured procedural language like Ada, or Pascal.

The Prolog used in this implementation was Edinburgh Prolog, and 'pure' Prolog, i.e. Prolog without 'not' and 'cut', was used to prototype the axiomatic specification. In essence, a program written in 'pure' Prolog is a specification of the solution to a problem. 'Pure' Prolog, as used in this animation is a sub-set of Prolog, and is totally declarative in nature, since it does not make use of the 'cut', which prevents 'backtracking', an essential feature of the language, when Prolog seeks to prove or disprove an assertion or 'goal'. The use of 'not' and 'cut' detract from the declarative nature of the language, being mainly concerned with efficiency.

9.2.1 Strategy for testing

The data structure underlying the 'Book' part of the ADT Maintenance_History, is a Knuth Binary Search Tree (BST), and that underlying the Anthology is a linked-list. The linked-list was tested first, then the Knuth BST.

Axioms can be viewed as a tool for refining natural language. The natural language description of a specification, without recourse to an axiomatic specification would probably suffer

from ambiguity and context sensitivity. However, a natural language description of an axiom is a 'refined' natural language description, which does not suffer the same disadvantages as unrefined natural language.

The Prolog code was written so that its natural language description matched the natural language description of the axiom.

An animation of a specification, using a prototyping language means that there is possibility that the specification may be transformed by the language being used to animate it, since the language is translating one representation into another. It was therefore important to write a procedure that mirrored, as closely as possible, the axiom being tested.

As an illustration of this point, consider the following stack axiom:

$$top(push(i, S)) = i$$

There are at least two ways to capture the semantics of this axiom using Prolog, written in a declarative manner.

---

top(X,Y,[X—Y]).

(where: X = top of stack, Y = rest of stack, [X—Y] = list, X = head of list.)

In this example the predicate 'top' has an arity of three, i.e. it has three arguments.

A natural language description of this predicate is: X is the item at the top of a stack, the remainder of the stack being Y, if X is the head of a list, whose tail is Y.

Consider the following stack of integers:

Stack = [3,5,1]

The prolog query and the response are shown below:

— ?- top(X,Y,[3,5,1]). (where: X = top of stack, Y = rest of stack, [3,5,1] = stack)

X = 3

Y = [5,1]

Another prolog procedure which accomplishes the same thing is shown below:

---

top(X,S) :- push(X,S,S1).

push(X,Y,[X--Y]).

In this example the predicate 'top' has an arity of two, i.e. it only has two arguments.

The prolog query and the response are shown below:

— ?- top(X,[3,5,1]). (where: X = top of stack, [3,5,1] = stack)

X = 3

An algebraic axiom is expressed as one operation applied to the *result* of another, and

while the first prolog procedure *implicitly* suggests that the stack has been produced from a push operation, the second prolog procedure *explicitly* states this. This second example also illustrates how this style of capturing the semantics allows an abstraction to a higher level, since it permits a query which returns only the item of interest, the item at the top of the stack.

9.2.2 The operations

The operation was deemed to be correctly specified if the Prolog translation of the axiom achieved its objective, i.e. if the actual output was the same as the expected output, defined by the natural language description of the axiom, *and* the operation preserved the strict hierarchy of the Knuth Binary Search Tree (BST).

The Prolog procedures used to perform operations on the Knuth BST, i.e. to animate the specification, were assigned the same names as the axioms being prototyped, to avoid confusion, and can be classified into two distinct types, primitive or complex. Complex operations are generalisations of primitives, operating on sets of nodes instead of on single nodes, as primitives do. In addition utility operations were written to provide input/output routines, and a means of inputting the data had to be devised, so that the information concerning the data structure could be manipulated by Prolog procedures. Often these operations were, of necessity, procedural in nature, but did not compromise the validity of the animation, since they were used for reading from and writing to files, including standard

input and output, and for formatting these data streams.

The axioms tested first were those corresponding to primitive and utility operations, since these underly the complex operations. These operations were Parent, First_child, Next_sibling, Previous_sibling. This in turn meant that some way had to be found of giving meaning to the infix operators 'greater' and 'less' since the normal integer comparison provided by Prolog's built-in operators had no validity, for node keys of the type a.b.c, where a,b,c are integers.

The operations can also be classified according to their role concerning the data structure, i.e. whether they are used in connection with Initialisation, Re-arrangement or Navigation. Primitive operations are concerned with Initialisation, or Navigation, and complex operations are concerned with re-arrangement of the data structure. The operations for the ADT Anthology are shown below in Table 9.1, and the operations for the ADT Book are shown below in Table 9.2.

9.3 Summary

A rationale has been given for the choice of Prolog as the language for the implementation, and a strategy for testing the implementation has been devised. The importance of coding style has been discussed. The operations have been listed and the criteria given for a correct specification of an axiom.

Prolog offers little in the way of procedures to handle input and output, and, of necessity, the user-interface for the prototype is somewhat primitive, but this was not intended to be a model for the fully-fledged ISMSE. The user-interface would need to be much more sophisticated, to minimise cognitive overhead and is a research topic in its own right.

The primitive interface was sufficient for the author to prototype the specification of the documentation paradigm. Prolog is quite verbose in the way it outputs its results, and some kind of filter needs to be incorporated to reduce this, to make Prolog a better prototyping tool. Building the data structures using Prolog is cumbersome and involves much repetition, including temporary storage of the data structure in a file. The design of suitable macros could do much to alleviate this.

Anthology		
Predicate	**Arguments**	**Purpose**
assign	In_L.Key.Name.Out_L	Give name to member
create_Anthology	None	Create empty Anthlology
delete	In_L.Key.Out_L	Remove a member of Anthlogy
earliest	In_L. Key	Return earliest member
evaluate	In_L,Key	Display name of Book
insert	In_L.key.Out_L	Insert a book into Anthology
isin_Anthology	In_L.key	Test for membership of Anthology
latest	In_L.Key	Return latest version in Anthology
next	In_L.Key.Next_Key	Return the next version
position	In_L.Key.Number	Return version number
previous	In_L.Key.Prev_Key	Return previous version
write	In_L.	Display identifier of member
Abbreviations		
In_L	Input Anthlogy	
Out_L	Updated Anthlogy	
Key	Alphanumeric identifier	
Prev_Key	Previous Key	
Number	Ordinal position in Anthology	

Table 9.1: The operations for the ADT Anthology

Book		
Predicate	**Arguments**	**Purpose**
abstract	(In_T,Start,End,List)	Return portion of chapter
assign	(In_T,Key,data)	Give name to node
bookmark	(List,Item)	Mark last node perused
copy_subtree	(In_T,Key,Out_T)	Make copy of subtree
create_Book	None	Create empty Book
cross_ref	(In_T,Key,List)	List all nodes containing Keyword
equal	(T1,T2)	Test equality of portions of Book
evaluate	(In_T,Key)	Display contents of a node
first_Child	(In_T,Key)	Display subordinate node Key
graft	(In_T,Key,Subtree)	Insert subset of the Book
home	(In_T)	Return to title node
insert_node	(In_T,Key,Out_T)	Insert node into the Book
is_empty Book	(In_T)	Test for empty Book
is_in Book	(Key,In_T)	Test for presence of a node in Book
last_Sibling	(In_T,Key)	Return last node in a list
move	(In_T,Kp,Kg,Dir,Out_T)	Re-position portion of the Book
next_Sibling	(In_T,Key)	Return next node in list
parent	(In_T,Key)	Display superordinate node Key
path	(In_T,Key,List)	Display path from root to target node
prev_sib	(In_T,Key)	Return previous member of the list
prune	(In_T,Key,Out_T)	Remove portion of the Book
table	(In_T)	Display contents of Book
trail	(Key,List)	Display list of nodes visited
Abbreviations		
In_T	Input Book	
Out_T	Output Book	
Start	Commencement node for Search	
End	Termination node for Search	
Kp	Root of subtree to be pruned	
Kg	Node where Graft of pruned subtree is to occur	
Dir	Root of subtree grafted as First_Child or Next_Sibling	

Table 9.2: The operations for the ADT Book

Chapter 10

Evaluation of the Documentation Paradigm

10.1 Introduction

The preferred test of the utility of the documentation paradigm, to determine its efficacy, i.e. in terms of its performance and ease of use, would be its application to a maintenance project of reasonable size and complexity. Unfortunately, this is infeasible, for two main reasons. Firstly, the ISMSE which would host the documentation paradigm has not yet

been built, and in addition, the time scale appropriate to this thesis does not permit such an investigation. To provide an examination of the capabilities of the documentation paradigm, its analysis and evaluation rest instead on its application to the maintenance of 'pxr', a cross-referencer for the Pascal programming language, which is a component of an ISMSE's toolset. The documentation paradigm is used to store information about the cross-referencer, this information is produced by the cross-referencer itself. The documentation paradigm is a prototype, and as such its evaluation constitutes a feasibility study. The operations used to interrogate and manipulate the Maintenance History for the maintenance of the cross-referencer, are analysed, and the results of the analysis are used to extrapolate from this scenario, making an inference as to the probable functionality of the documentation paradigm, when used in the maintenance of a large, complex, software system.

10.2 Applying the documentation paradigm

10.2.1 Introduction

The database is the nucleus of an environment and is responsible to a large extent in shaping its character and functionality. This section demonstrates how the adoption of the documentation paradigm as the conceptual schema for the environment database will provide the ISMSE with some of those features of an environment which Magel [77] showed to be desirable. Later in this chapter a description is given of the support provided by the envi-

ronment for its underlying maintenance process model, this support being a prerequisite to the provision of these desirable features. In this section an analysis of the utility of the documentation paradigm is undertaken, using the operations carried out on the book structure during the maintenance of 'pxr', and from this base extrapolating to the maintenance of a large complex software system.

10.2.2 Placing the documentation paradigm in context

An anthology of hierarchical maintenance logs, i.e. books, comprises the 'Maintenance History' of the software, and forms the basis of a documentation paradigm for the ISMSE, the aim of this thesis. Each book of the anthology encompasses a version or variant of the software system, as shown below in Figure 10.1.

The technology underlying integrated software engineering support environments is not yet mature enough to permit their wide-scale use. Consequently, in the forseeable future much maintenance will be supported using toolkit environments, for example, Unix. Such environments do not make use of sophisticated database management systems, but instead rely on the host computer's filing system, which is usually a hierarchical one. To demonstrate the utility of the documentation paradigm the operations will be referred to a hierarchical file system and typical operating system commands used to manipulate the file structure, for example the addition and deletion of files, and the movement of files from one directory to another, the renaming of files, and so on. The node key gives the location of the file

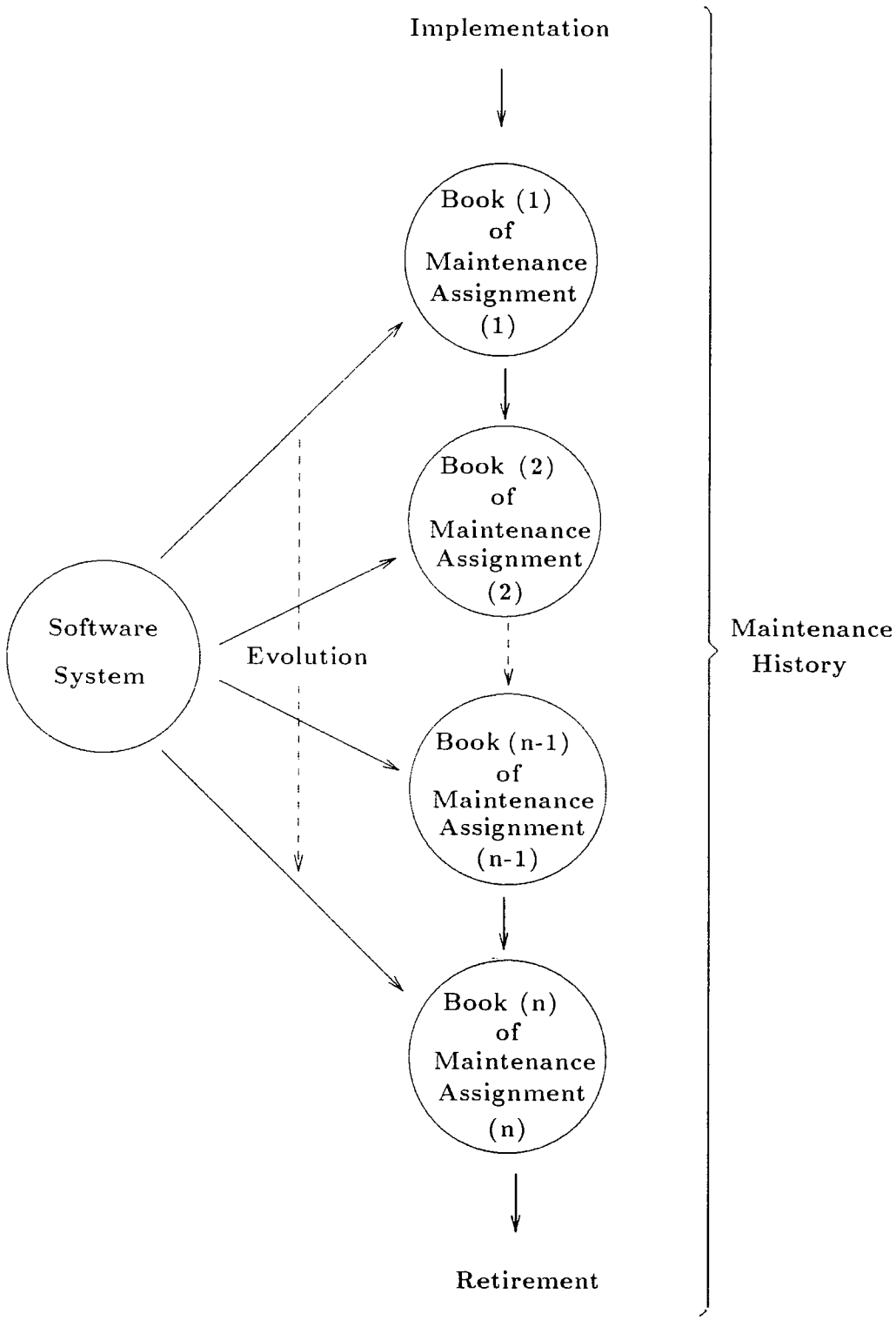


Figure 10.1: The Maintenance History of a software system

in the hierarchical file system, the length of the key indicating the level in the hierarchy corresponding to the file's location. This means that filenames can still be meaningful, the documentation paradigm serving as an indexing system for the filestore. The documentation paradigm serves as a conceptual schema for a database, which is based on the host system's filestore, the nodes of the Maintenance History corresponding to files. A well-established practice in software engineering is for a project to be mapped to a hierarchical directory structure, with each component of the project existing as a document. The Maintenance History corresponds to an open-ended project which lasts until the retirement of the software. The Maintenance History has been described as an Anthology of books, the structure of the Anthology is mirrored by having within the root directory of the hierarchical file system, a directory containing the first book of the Anthology, and another directory containing the remainder of the Anthology, which contains the next Book in the Anthology, and the remainder of the Anthology. This decomposition is repeated throughout the hierarchical directory structure, as shown in the diagram below, and implements a *list* of books, each book having a *tree* structure. Together the list and the tree comprise the ADT Maintenance_History. The position in the hierarchical file structure of each *book* is indicated by 'Level' in Figure 10.2 below, Level 0 corresponding to the root of the directory structure containing the first book of the Anthology.

```
Title              Anthology of pxr

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 0

 {book1            version 1.1
 Anthology'}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 1

 {book2            version 1.2
 Anthology''}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 2

 {book3            version 1.3
 Anthology'''}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 3


...


 bookN             version n
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level n
```

Figure 10.2: Project structure of maintenance of pxr

The hierarchical structure of the book maps to the hierarchical file structure, each component of the book structure corresponding to a file in the file system. The node-key 'title' is the root of the hierarchical directory structure, which contains the book. The book documenting the maintenance of 'pxr' is shown below in Figure 10.3 below, mapped to a hierarchical file structure. The position in the hierarchical file structure of each book *component* is indicated by 'Level' in the figure, Level 0 corresponding to the root of the directory containing the title of a book. The book is shown incomplete for reasons of clarity.

In the root directory called title, there are the chapter directories. Contained in each chapter directory, is a file called chapter and a directory called section. The file called chapter will contain an introduction to the chapter. The directory called section contains the section directories, each section directory contains one file which contains information, and a directory called sub-section. This decomposition is repeated throughout the hierarchical directory structure, to reflect the structure of the book.

```
Title               Version 1.1 pxr

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 0

 chapter 1              The change requests

 chapter 2              Understanding the software

...

{chapter 4              Revalidation of the software
 section}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 1

 section 4.1            Integration testing


{section 4.2            Regression testing
 subsection}

...

 section 4.4            Documentation of changes made to 'pxr' source code

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 2

{subsection 4.2.1       Test Cases
 subsubsection}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 3

 subsubsection 4.2.1.1 Test Results

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Level 4
```

Figure 10.3: Book structure of a version of pxr

Much of software maintenance is concerned with the enhancement of software, which is software development. The documentation paradigm is compatible with the Software Development Life-Cycle model, since software maintenance is an iteration of this Life-Cycle. There is a relation between the hierarchical data model of the documentation paradigm and the Software Life-Cycle, in that the early stages of the Software Life-Cycle map to high-level *abstractions* (or views) of the software system and the later stages of design, testing and coding map to lower-level abstractions. The document set is partitioned, so that, for instance, source code modules are placed in a separate directory, from the documentation concerning these modules. In addition, the organisation of the project team is a hierarchical one, the constituent parts of the team hierarchy being delegated responsibility for a particular portion of the document hierarchy, which makes possible a simple mapping. The maintenance team will be available to work as individuals on a particular chapter of the log and their efforts pooled, using operations to assemble the components of the chapter into its final form. Overall, the documentation paradigm provides support for *grouping of resources*.

The Maintenance History is a hierarchical archive of information, and understanding is partly concerned with retrieval of information from this archive. The granularity of the objects to be stored in the information structure will vary greatly, and the relationships between them may be complex: to cope with this complexity requires that support is provided for abstraction. Information is usually documented at various levels of abstraction, and program understanding depends on access to up-to-date information concerning the source code. Inline comments in the source code (known as internal documentation) are usually too low-level, being concerned with descriptions of algorithms and properties of data items. There

are two avenues of approach for accessing information concerning the source code. Static and dynamic program-analysis tools can be used, which operate on the source code itself, which means that the output of these tools has to be interpreted by the maintainer. In view of this it is advantageous to be able to have access to the *high-level* information from its documentation concerning the system's function, how it functions, including the system components needed to express its functionality. The documentation paradigm supports this strategy.

Detailed information can act as a barrier to understanding. This link between understanding and abstraction is well-accepted. A hierarchical organisation of information relates to the human mind's problem-solving capability - that of rough formulation of a solution to a problem, followed by stepwise refinement of the model through hypothesis and the testing of this hypothesis, resulting in a hierarchical decomposition. This often requires the design of an experiment to be used as a tool for testing the hypothesis, which can also be stored in the information structure. A hierarchical information structure for the documentation of the maintenance of software enables the organisation of levels of abstraction, which will make understanding of the program easier for *future* maintainers. Incorporation of the strategy of abstraction within the overall hierarchical structure of a book format of the documentation is consistent and easily achieved. The documentation paradigm supports the recording of information at different levels of abstraction, by the provision of a hierarchical information structure, i.e. an acyclic directed graph, the table of contents operation providing a *conceptual map* of the documentation concerning the maintenance of a software system. This operation can also serve as a management tool, when monitoring the maintenance process,

the table of contents at any one time displays the structure of the book and so can show the current position with regard to the stage of completion of the maintenance assignment. The large body of documentation associated with a software system often possesses a strong amorphous character, containing information associated with every phase of the Software Development Life Cycle. The documents comprising the documentation of a software system are written in a diversity of styles and formats, and it is therefore to be expected that difficulties will arise when attempting to gain an understanding of the system from a study of the documentation. Failure to bring this documentation under control by imposing some structure on it, when performing maintenance on the system, will ensure that the existing situation is perpetuated.

The utility of having a large body of information concerning a software system can only be realised if a strategy exists for the subsequent rapid retrieval of information from the information-structure used to store these diverse types of information. Without this rapid retrieval the process of reaching an understanding of the software system will be hindered, increasing the time taken to achieve the necessary understanding for modification of the program, and its associated documentation to begin, thus reducing the productivity of a maintenance organisation. Binary search is possible when the book structure is that of an ordered binary tree, making possible the desired rapid retrieval of information.

An information structure is designed to record the relationships between pieces of information and to provide ways of using, changing, and managing it. The adoption of a book format for the information structure ensures that the documentation of a software system has a standard organisation, the importance of which has been shown by Selig [115]. Anyone

involved with the software system can find an item of information, without needing to learn any new concepts associated with an *unfamiliar* information-structure, i.e. the book format offers an information-structure which is natural and totally familiar, making it easier to gain an understanding of large complex system. This model of information-presentation has stood the test of time, showing that it possesses great utility and durability; it provides several ways to access the information held there, e.g. table of contents, index, glossary, chapter, section, and provides facilities for cross-referencing.

10.2.3 The maintenance of a Pascal cross-referencer 'pxr'

10.2.3.1 Introduction

The Pascal cross referencer 'pxr' resembles the 'front-end' of a compiler, lacking only its code generation capability, and contains lexical analysis, syntax analysis, and symbol table manipulation routines. From standard input 'pxr' reads in a Pascal program and generates either an alphabetic or structural cross-reference listing, of the identifiers used in the program, along with the lines on which they appear. The cross-referencer consists of approximately 8,000 lines of Pascal code in thirty modules.

Cross-referencers can output copious amounts of information, which often means that their contribution to understanding a program is less than predicted, because of the time-consuming and error-prone task of sifting through this information. For this reason 'pxr' offers options

as regards the type of output listing. Pascal is a block-structured language, and the user can opt for a structured listing, which alphabetically lists identifiers within the scope of each block; the detail of the listing corresponding to *terse*, *full*, or *intermediate*, reflecting the degree of detail given for each identifier. In the *terse* option no distinction is made between appearances of each identifier, whereas, when using the full option, information is given as to where the identifier is used, set, and called. The user enters a command with the appropriate t (*terse*), f (*full*), or i (*intermediate*), flag from within an operating system shell, and this command causes 'pxr' to output the desired listing.

10.2.4 Production of a Maintenance History for pxr

Operations on the Maintenance History can be classified as primitive or complex. Primitive operations often are concerned with single nodes, so that in the context of the file system the minimum granularity of the operand for the operation is the file. Complex operations are generalisations of primitives, operating on *sets* of nodes instead of single nodes, corresponding to a subtree, or even the whole tree, i.e. the operand is an *object*. In the context of the file system this corresponds to a collection of files, perhaps a directory.

The complex operations express the utility of the documentation paradigm, and are at a higher semantic level than some of the primitive operations which are used to create and build the Maintenance History; these primitive operations contribute little to the illustration of the utility of the paradigm, so they have not been shown. Some of the complex operations

possess the ability to retrieve information in a way specified by the maintainer, and provide the maintainer with different views of the software, through the ability to parameterise these operations, which provides a powerful tool for procedural abstraction. Each operation illustrated below is used to describe one of the benefits of using the documentation paradigm in the maintenance of a large software system, these operations together contribute to the definition of its overall semantics.

The maintenance model derived in chapter 2 was used in the maintenance of 'pxr', and is reproduced below.

1. Verification of the need for maintenance
2. Understanding
3. Modification
4. Revalidation

It was mentioned earlier in this chapter that the environment should provide good support for its underlying maintenance process model, the hierarchical topology of the book ensures that the maintenance model adopted for the environment is supported, the book structure being partitioned into **contexts**, these contexts corresponding to the phases of the maintenance model.

This partitioning is achieved by using the notion of chapters. Within each chapter the information is again partitioned, into sections and subsections, paragraphs, and subparagraphs

as in a book, e.g. different views of the software can be presented as sections within chapters of the book of the maintenance assignment, each component of the book structure having an information type associated with it.

Adopting this approach means that the documentation of the maintenance activity is a 'by-product' of that activity. When a future maintenance team comes to read the book concerning the past maintenance performed on the software, the use of the change request as a template for the structure of the book provides a useful beginning to understanding how the maintenance performed on the system relates to that change request.

The operations which define the semantics of the ADT Maintenance_History are now illustrated by reference to the maintenance of 'pxr'. The Maintenance History, was produced using the operations listed in chapter 9. The operations undertaken during the maintenance of 'pxr' are used in the execution of a realistic selection of tasks, from each of the phases of the maintenance model. The documentation paradigm is not intended to be used in the *performance* of maintenance, nor is it intended that it is to be used in isolation in recording, monitoring and managing maintenance. It is envisaged that it will rely heavily on supporting automation such as configuration management tools, e.g. RCS [123] and Make [38], and a text editor, if it is to achieve the desired functionality.

10.2.4.1 Chapter 1 - Verification of the need for maintenance

Problem reports revealed deficiencies in the operation of pxr, and evaluation of these reports gave rise to change requests, which identify those features of the cross-referencer which make

maintenance necessary: these requests appear below.

1. Output on screen is confusing due to output of tab characters, and sometimes the screen is cleared inappropriately.

2. The program doesn't distinguish between Var, Value, Procedure and Function parameters.

3. Self-referencing types are not output.

4. The cross reference listing should not include standard Pascal types.

5. The structured listing has not been implemented.

6. The terse/intermediate listings do not work as specified.

7. The full alphabetic listing is not implemented.

Since 'pxr' is new software it has no Maintenance History and so no search is necessary to establish whether the change requests have been previously satisfied. If a Maintenance History did exist for 'pxr' then a content search could have been used to verify the need for maintenance, by searching the Anthology for a keyword contained in the change request, using the **cross_reference** operation. The utility of this operation is illustrated later in the ordering of the change requests.

Before embarking on a search of the documentation, to find which parts of the program require attention, the requests were submitted to a change-control authority and following its approval, access to the source code was given.

After creating the Anthology, using the **create_Anthology** operation, and within it the book, using the **create_Book** operation, the name and the version of the software system being created was adopted as the title of the book. A skeleton book was constructed, comprising six chapters, using the **insert** operation. The first four chapters concern the four phases of the maintenance model, chapter five summarises the maintenance assignment, and chapter six functions as a 'Scratchpad', so that the maintenance team have somewhere to test ideas concerning the working of the software, before writing up the appropriate component of the book. This Scratchpad can be partitioned to reflect the structure of the maintenance team. At this stage the **table** operation shows the book structure to be as shown in Figure 10.4 below.

```
Title  Version 1.1 pxr
1 The Change requests
2 Understanding the software
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
```

Figure 10.4: The Maintenance History of pxr

The change requests were then entered into the book structure as they appeared on the change request document, using the **insert** operation, to give the structure as shown below, in Figure 10.5 below.

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Screen output confusing
  1.2 Program doesn't distinguish types of parameter
  1.3 Self-referencing types not output
  1.4 Output of standard Pascal types not required
  1.5 Structured listing not yet implemented
  1.6 Terse/intermediate listings do not work as specified
  1.7 Full alphabetic listing not yet implemented
2 Understanding the software
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
```

Figure 10.5: The Maintenance History of pxr

The maintenance team assigned a priority to each request, firstly according to the type of maintenance involved, corrective maintenance having the greatest priority, and secondly according to the module currently being maintained, as will be illustrated below. Corrective maintenance was required to enable 'pxr' to distinguish between types of parameter, and to output self-referencing types. Accordingly, in descending order, the new priority becomes as shown in Figure 10.6 below.

This new ordering was copied to the scratchpad area of the book, using an iteration of the

```
1. Program doesn't distinguish  types of parameter
2. Self-referencing types not output
3. Screen output confusing
4. Output of standard Pascal types not required
5. Structured listing not yet implemented
6. Terse/intermediate listings do not work as specified
7. Full alphabetic listing not yet implemented
```

Figure 10.6: New priority for change requests

copy_tree operation, and the new book structure was as shown by the **table** operation in

Figure 10.7 below.

The five remaining change requests were concerned with perfective maintenance, and since

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Screen output confusing
  1.2 Program doesn't distinguish types of parameter
  1.3 Self-referencing types not output
  1.4 Output of standard Pascal types not required
  1.5 Structured listing not yet implemented
  1.6 Terse/intermediate listings do not work as specified
  1.7 Full alphabetic listing not yet implemented
2 Understanding the software
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
    6.1 Program doesn't distinguish types of parameter
    6.2 Self-referencing types not output
```

Figure 10.7: The Maintenance History of pxr

it was not yet known which modules of 'pxr' were involved in the change requests, the remainder of the ordering was postponed until the next phase of maintenance, i.e. the understanding phase.

10.2.4.2 Chapter 2 - Understanding

The external documentation produced during the development of the software, was used to further a global understanding of the program, relevant source code modules associated with the change requests were identified, and were entered into the Scratchpad area of the book structure, using the **insert** operation. The **table** operation shows the new state of the

book in Figure 10.8 below. Since the modules concerned with all the change requests were

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Screen output confusing
  1.2 Program doesn't distinguish types of parameter
  1.3 Self-referencing types not output
  1.4 Output of standard Pascal types not required
  1.5 Structured listing not yet implemented
  1.6 Terse/intermediate listings do not work as specified
  1.7 Full alphabetic listing not yet implemented
2 Understanding the software
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
    6.1 Distinguishing parameter types
    6.2 Self-referencing types not output
  6.2 Modules associated with change requests
    6.2.1 Distinguishing  parameter types: paramlist.p, print.p, symbol.p
    6.2.2 Self-referencing types: readtype.p
    6.2.3 Screen output: gettoken.p
    6.2.4 Standard Pascal types: symbol.p
    6.2.5 Structured listing: print.p
    6.2.6 Terse/intermediate listings: print.p
    6.2.7 Full alphabetic listing: arguments.p, print.p
```

Figure 10.8: The Maintenance History of pxr

now known it is possible to complete the ordering of the change requests, for the purpose of drawing up a maintenance plan, by examining which of the five remaining change requests are concerned with the modules involved with the **first** item in the change request queue, i.e. distinguishing parameter types.

The **cross_reference** operation was used to find which of the remaining change requests were concerned with the paramlist.p, print.p, and symbol.p modules.

The operation **Cross-reference** implements a content search and produces a list of components of the book structure which contain a common word in the data associated with the component. These components are concerned with the same topic, and thus are grouped together, for perusal, giving the opportunity for 'cross-fertilization', acting as a catalyst for understanding.

The output from the cross reference operation for each of the modules in the first item of the change request document was entered into the Scratchpad area of the book for each module, using the **insert** operation, and according to the number of occurrences the **insert** operation was used to insert the remaining change requests into their appropriate position in the change request queue.

The **graft** operation unifies two subtrees into a single tree. The node where the graft is to occur and the root of the grafted subtree are specified. Graft can be used to unify the contributions of the maintenance team, within the Scratchpad.

The state of the book is shown in Figure 10.9 below using the **table** operation.

The original order of the change requests in chapter 1 was then deleted using the **prune** operation, and the final order of the change requests was transferred from the Scratchpad chapter using the **move** operation.

Attention was now focused on understanding the software, which is in the domain of chapter 2. The section 2.1 'Global Understanding' was inserted using the **insert** operation, then subsections 6.2.1 to 6.2.7 inclusive were transferred to section 2.1 of chapter 2 using the

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Screen output confusing
  1.2 Program doesn't distinguish types of parameter
  1.3 Self-referencing types not output
  1.4 Output of standard Pascal types not required
  1.5 Structured listing not yet implemented
  1.6 Terse/intermediate listings do not work as specified
  1.7 Full alphabetic listing not yet implemented
2 Understanding the software
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
    6.1 Program doesn't distinguish types of parameter
    6.2 Self-referencing types not output
    6.3 Structured listing not yet implemented
    6.4 Terse/intermediate listings do not work as specified
    6.5 Full alphabetic listing not yet implemented
    6.6 Output of standard Pascal types not required
    6.7 Screen output confusing
  6.2 Modules associated with each change request
    6.2.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    6.2.2 Self-referencing types not output: readtype.p
    6.2.3 Screen output confusing: gettoken.p
    6.2.4 Output of standard Pascal types: symbol.p
    6.2.5 Structured listing: print.p
    6.2.6 Terse/intermediate listings do not work: print.p
    6.2.7 Full alphabetic listing not yet implemented: arguments.p, print.p
  6.3 Requests linked with paramlist.p, print.p, symbol.p
    6.3.1 Requests linked with paramlist.p
      6.3.1.1 Distinguishing  parameter types: paramlist.p, print.p, symbol.p
    6.3.2 Requests linked with print.p
      6.3.2.1 Distinguishing  parameter types: paramlist.p, print.p, symbol.p
      6.3.2.2 Structured listing: print.p
      6.3.2.3 Terse/intermediate listings: print.p
      6.3.2.4 Full alphabetic listing: arguments.p, print.p
    6.3.3 Requests linked with symbol.p
      6.3.3.1 Distinguishing  parameter types: paramlist.p, print.p, symbol.p
      6.3.3.2 Standard Pascal types: symbol.p
```

Figure 10.9: The Maintenance History of pxr

move operation. The **table** operation now reveals the structure of the book to be as shown in Figure 10.10 below. A local understanding of the program was achieved by a perusal

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Program doesn't distinguish types of parameter
  1.2 Self-referencing types not output
  1.3 Structured listing not yet implemented
  1.4 Terse/intermediate listings do not work as specified
  1.5 Full alphabetic listing not yet implemented
  1.6 Output of standard Pascal types not required
  1.7 Screen output confusing
2 Understanding the software
  2.1 Global Understanding
    2.1.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    2.1.2 Self-referencing types not output: readtype.p
    2.1.3 Structured listing: print.p
    2.1.4 Terse/intermediate listings do not work: print.p
    2.1.5 Full alphabetic listing not yet implemented: arguments.p, print.p
    2.1.6 Output of standard Pascal types: symbol.p
    2.1.7 Screen output confusing: gettoken.p
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
  6.2 Modules associated with each change request
  6.3 Requests linked with paramlist.p, print.p, symbol.p
```

Figure 10.10: The Maintenance History of pxr

of the change request and a detailed study of the relevant modules of the source code, in conjunction with a study of the output from 'pxr', to obtain a design for the changes to the source code, in preparation for the next phase of maintenance, i.e. the modification phase. The **insert** operation was used to insert section 2.2 'Local Understanding' into Chapter 2 and also the subsections detailing the function of each of the modules involved, as shown in Figure 10.11 below.

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Program doesn't distinguish types of parameter
  1.2 Self-referencing types not output
  1.3 Structured listing not yet implemented
  1.4 Terse/intermediate listings do not work as specified
  1.5 Full alphabetic listing not yet implemented
  1.6 Output of standard Pascal types not required
  1.7 Screen output confusing
2 Understanding the software
  2.1 Global Understanding from external documentation
    2.1.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    2.1.2 Self-referencing types not output: readtype.p
    2.1.3 Structured listing: print.p
    2.1.4 Terse/intermediate listings do not work: print.p
    2.1.5 Full alphabetic listing not yet implemented: arguments.p, print.p
    2.1.6 Output of standard Pascal types: symbol.p
    2.1.7 Screen output confusing: gettoken.p
  2.2 Local Understanding
    2.2.1 gettoken.p: lexical analysis of command line.
    2.2.2 paramlist.p: parsing  formal and actual parameters.
    2.2.3 print.p: printing  cross-reference listing.
    2.2.4 symbol.p:  processing  symbol table.
    2.2.5 readtype.p: parsing  declarations in the program.
    2.2.6 arguments.p: processing arguments in the command line parsing.
3 Modification of the software
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
  6.2 Modules associated with each change request
  6.3 Requests linked with paramlist.p, print.p, symbol.p
```

Figure 10.11: The Maintenance History of pxr

10.2.4.3 Chapter 3 - Modification

The changes to the source code modules to implement the change requests were designed for each module, and inserted as sub-sections into chapter 3 of the book using the **insert** operation, to produce the book structure shown in Figure 10.12 below by the **table** operation. The source code was amended as indicated above to ensure that the cross-referencer performed according to specification. Testing the changes made to the software was then carried out in the revalidation phase.

10.2.4.4 Chapter 4 - Revalidation

Integration testing was carried out using a test file which contained all the features of Pascal and this testing confirmed that the new version of pxr functioned as required by the change request. Regression testing was carried out using a test suite to verify that the changes made had no adverse side-effects on the program. The test file, the test suite and the test results were included in the chapter using the **insert** operation. The amended book structure is shown below in Figure 10.13 below, and was produced using the **table** operation. The external documentation relating to the program was updated to reflect the changes made. Approval from the change control authority was followed by incorporation of the modified modules into a new version of pxr, using RCS [123] and Make [38]. The support offered by the documentation paradigm for *extensibility* is now illustrated by reference to the support offered for documentation and configuration management.

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Program doesn't distinguish types of parameter
  1.2 Self-referencing types not output
  1.3 Structured listing not yet implemented
  1.4 Terse/intermediate listings do not work as specified
  1.5 Full alphabetic listing not yet implemented
  1.6 Output of standard Pascal types not required
  1.7 Screen output confusing
2 Understanding the software
  2.1 Global Understanding from external documentation
    2.1.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    2.1.2 Self-referencing types not output: readtype.p
    2.1.3 Structured listing: print.p
    2.1.4 Terse/intermediate listings do not work: print.p
    2.1.5 Full alphabetic listing not yet implemented: arguments.p, print.p
    2.1.6 Output of standard Pascal types: symbol.p
    2.1.7 Screen output confusing: gettoken.p
  2.2 Local Understanding
    2.2.1 gettoken.p: lexical analysis of command line.
    2.2.2 paramlist.p: parsing  formal and actual parameters.
    2.2.3 print.p: printing  cross-reference listing.
    2.2.4 symbol.p:  processing  symbol table.
    2.2.5 readtype.p: parsing  declarations in the program.
    2.2.6 arguments.p: processing arguments in the command line parsing.
3 Modification of the software
  3.1 arguments.p: code  added to parse  'f' and 'F' options
  3.2 gettoken.p: no tab characters output, call to 'page' removed
  3.3 paramlist.p: parameters distinguished
  3.4 print.p: parameters distinguished; parameter information updated;
      alphabetic and intermediate/terse listings implemented
  3.5 readtype.p: type-name inserted into symbol table before processing
  3.6 symbol.p: parameters distinguished; no output of standard names
4 Revalidation of the software
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
  6.2 Modules associated with each change request
  6.3 Requests linked with paramlist.p, print.p, symbol.p
```

Figure 10.12: The Maintenance History of pxr

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Program doesn't distinguish types of parameter
  1.2 Self-referencing types not output
  1.3 Structured listing not yet implemented
  1.4 Terse/intermediate listings do not work as specified
  1.5 Full alphabetic listing not yet implemented
  1.6 Output of standard Pascal types not required
  1.7 Screen output confusing
2 Understanding the software
  2.1 Global Understanding from external documentation
    2.1.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    2.1.2 Self-referencing types not output: readtype.p
    2.1.3 Structured listing: print.p
    2.1.4 Terse/intermediate listings do not work: print.p
    2.1.5 Full alphabetic listing not yet implemented: arguments.p, print.p
    2.1.6 Output of standard Pascal types: symbol.p
    2.1.7 Screen output confusing: gettoken.p
  2.2 Local Understanding
    2.2.1 gettoken.p: lexical analysis of command line.
    2.2.2 paramlist.p: parsing  formal and actual parameters.
    2.2.3 print.p: printing  cross-reference listing.
    2.2.4 symbol.p:  processing   symbol table.
    2.2.5 readtype.p: parsing  declarations in the program.
    2.2.6 arguments.p: processing arguments in the command line parsing.
3 Modification of the software
  3.1 arguments.p: code  added to parse  'f' and 'F' options
  3.2 gettoken.p: no tab characters output, call to 'page' removed
  3.3 paramlist.p: parameters distinguished
  3.4 print.p: parameters distinguished; parameter information updated;
      alphabetic and intermediate/terse listings implemented
  3.5 readtype.p: type-name inserted into symbol table before processing
  3.6 symbol.p: parameters distinguished; no output of standard names
4 Revalidation of the software
  4.1 Integration testing
  4.2 Regression testing
    4.2.1 Test Cases
      4.2.1.1 Test Results
5 Executive Summary
6 Scratchpad
  6.1 Ordering the change requests
  6.2 Modules associated with each change request
  6.3 Requests linked with paramlist.p, print.p, symbol.p
```

Figure 10.13: The Maintenance History of pxr

Support for completeness and consistency of documentation

Primarily, the documentation paradigm is a tool to assist in maintaining the completeness and consistency of the documentation of maintenance of large, complex, software systems. Consequently, this support is analysed in the context of the revalidation phase of software maintenance, but it is equally relevant to the understanding and modification phases of maintenance.

The Maintenance History is an ADT which is made up of a list of hierarchical structures, i.e. books. This hierarchical concept means that the documentation paradigm supports the concept of *extensibility*, one of the desirable characteristics of environments listed by Magel [77].

An illustration of this extensibility is the provision of an enforced linkage between the source code and its external documentation. The operation **equal** used in the maintenance of 'pxr' implements a *structure* search and tests two trees for equality: the trees may be subtrees within the structure of the Book, or the trees may be Books themselves. In the context of extensibility, the trees are the structural hierarchy of the source code and the structural hierarchy of the documentation of this source code. The test for equality is in respect of their structure and the values of their node keys. A structured program has its modules, procedures, functions, generically known as components, arranged in a hierarchical manner, as shown in Figure 10.14 below. The letters A,B etc represent the component identifiers. It is good programming practice to give these component identifiers meaningful names, but these names alone do not reflect a hierarchical program structure. To make the hierarchy explicit,

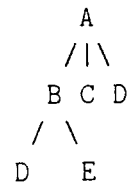


Figure 10.14: Module hierarchy of a structured program

the component identifier should carry two 'labels', a prefix and a suffix. The prefix reflects the place of the component in the hierarchy of the source code, and a numerical suffix indicates the version number of the component, incremented each time the component is modified. The modified hierarchical program component identifiers would then appear as shown in Figure 10.15 below. The hierarchical structure of the program is now explicit, and the place

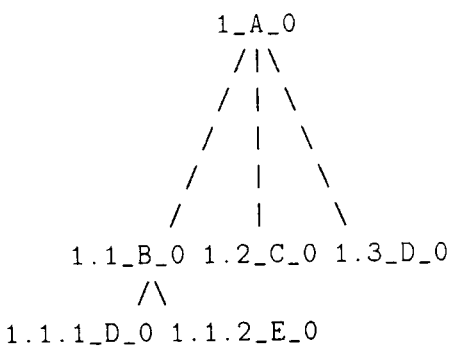


Figure 10.15: Modified module hierarchy of a structured program

of a component within the hierarchy can be seen from its identifier. The documentation of source code can mirror the hierarchy of the program, the problem of *existing* code without these prefixes and suffixes could be solved by interfacing with a cross-referencer, the source code and the documentation file could then be revised to carry these identifiers. Manually scanning the software would be less of a chore, through enhanced readability of the source code, and of the cross-reference output.

Documentation is more likely to be done well if there exists an easy means of updating it, and if there exists an audit process to check whether documentation has been written in compliance with the standards in force at the time of writing. This is a vital issue, since failure to maintain documentation devalues the software, which is a capital asset. Update of documentation is required to reflect changes made to the source code. An organisation may decide that 'free-format' documentation is allowable, but auditing procedures may find it difficult to verify this type of documentation, so 'form-fill' may be the standard method for documentation of maintenance. Whatever method is used, the documentation paradigm is flexible and abstracts away from this low-level aspect of documentation.

Two types of modification to the source code can be envisaged, the structure of the hierarchy could be changed, e.g. by the addition or removal of components, which concerns the role of the prefix, or internal modifications could be made to a component, which concerns the role of the suffix. Suppose another module had been inserted between modules whose identifier prefixes are 1.1.1 and 1.1.2, respectively. This inserted module would then have its identifier prefix as 1.1.2 and the module with a prefix of 1.1.2 would have its prefix incremented to 1.1.3. Each time a source code module is modified, its identifier suffix is incremented by one, so that if two module identifiers only differ in the value of their suffixes, then the module identifier with the greater value of its suffix is the later version.

All components could carry the suffix value 0 initially and this could be incremented by one each time an internal modification is carried out: the components in the documentation file would be likewise incremented. A comparison of the modified source code structure with the corresponding documentation structure indicates where in the documentation structure

the maintainer needs to insert documentation concerning the change made to the source code, i.e. the place(s) in the structure of the documentation where it needs to be updated is given by a failure in a pattern match, either because the remainder of the identifier does not match, e.g. 1.1.2_identifier_X with 1.1.2_identifier_Y, or because the sequence of numbers does not match, e.g. 1.1.3 follows 1.1.2 instead of the expected 1.2 as in the original hierarchy. Correlation of the identifiers of the modules in the source code hierarchy and in the documentation hierarchy is made possible by the unique prefix-suffix combination of the identifiers, and this indicates whether the documentation has been updated to reflect any modifications to the structure of the source code. Each time the documentation file is updated it is kept as a new version with the name of the person(s) responsible for its last update, so a check can be made on the quality of the documentation.

Both the source code and its associated documentation concerning the maintenance assignment can be stored within the Book, as subtrees within its structure, providing an efficient means of testing the equality of these subtrees, as regards their structures and the values of the identifiers within those structures.

Applying the documentation paradigm to the maintenance of large software systems means that the Maintenance History can be used as a means of revising the original documentation without actually updating all its constituent parts, i.e. the paradigm provides a means of traceability between maintenance assignments. For example, instead of having to find every place in the documentation that makes reference to a variable, and listing the changes made, the use of new variables and the discontinuance of other variables could be kept centrally in a data dictionary, which is inherited by successive maintenance books: the same can

be done for the call-graph structure. This method functions in much the same way as an 'errata' insertion in a book. The data dictionary and call-graph structure will be updated and annotated to show which changes have taken place, from their 'baselines' i.e. as they were at the commencement of the maintenance assignment.

Support for configuration management

Another example of the support offered by the documentation paradigm for extensibility is the domain of configuration management. There are two strands to configuration management, that of the software system being maintained and that of the project structure itself.

A particular version of source code is comprised of particular versions of the source code modules. The suffix in the module identifier gives the version of the module to be included in a configuration of the software for a 'customer', each different configuration can be represented by a book in the Maintenance History. The figure shows all the values of the module suffixes as zero, this book would represent a 'base release' of the software.

The adoption of a hierarchical project structure means that the *relationship* between documents is made *explicit* since, when stored as the components of a book, for example, the relationship may be that of chapter and section, or section and subsection. Establishing a relationship between documents is essential for configuration management, in this respect the documentation paradigm is an *expression* of the maintenance model. Although the maintenance model is capable of evolution, this must be carefully controlled since it both

provides a standard configuration for the project structure and a familiar organisation of information so that members of the maintenance organisation know where to look in the book for particular kinds of information. In addition the project structure makes possible allocation of duties to the members of the project team.

The **insert** operation was used to include the changes made to the documentation, and the configuration of the new version of the software: the Scratchpad was removed from the book using the **prune** operation, to give the book structure as shown by the operation in Figure 10.16 below.

10.2.4.5 Chapter 5 - Executive Summary

After the documentation of the software an Executive Summary needs to be prepared. The importance of providing this type of information flow is that the management team have access to an overview of the maintenance assignment, which can aid them in their strategic planning. An example of the type of information included in the Executive Summary would be a statistical evaluation of the types of source code components which are most often changed (corrective maintenance) and would indicate the type of code that most often causes problems, selecting candidate modules for preventive (scheduled) maintenance. This information could be obtained from the suffixes of the source code component identifiers, using the **cross-reference** operation. Information may also be included in the Executive Summary to enable the management team to evaluate the toolset of the environment.

```
Title  Version 1.1 pxr
1 The Change requests
  1.1 Program doesn't distinguish types of parameter
  1.2 Self-referencing types not output
  1.3 Structured listing not yet implemented
  1.4 Terse/intermediate listings do not work as specified
  1.5 Full alphabetic listing not yet implemented
  1.6 Output of standard Pascal types not required
  1.7 Screen output confusing
2 Understanding the software
  2.1 Global Understanding from external documentation
    2.1.1 Program doesn't distinguish types of parameter: paramlist.p,
          print.p, symbol.p
    2.1.2 Self-referencing types not output: readtype.p
    2.1.3 Structured listing: print.p
    2.1.4 Terse/intermediate listings do not work: print.p
    2.1.5 Full alphabetic listing not yet implemented: arguments.p, print.p
    2.1.6 Output of standard Pascal types: symbol.p
    2.1.7 Screen output confusing: gettoken.p
  2.2 Local Understanding
    2.2.1 gettoken.p: lexical analysis of command line.
    2.2.2 paramlist.p: parsing  formal and actual parameters.
    2.2.3 print.p: printing  cross-reference listing.
    2.2.4 symbol.p:  processing   symbol table.
    2.2.5 readtype.p: parsing  declarations in the program.
    2.2.6 arguments.p: processing arguments in the command line parsing.
3 Modification of the software
  3.1 arguments.p: code  added to parse  'f' and 'F' options
  3.2 gettoken.p: no tab characters output, call to 'page' removed
  3.3 paramlist.p: parameters distinguished
  3.4 print.p: parameters distinguished; parameter information updated;
      alphabetic and intermediate/terse listings implemented
  3.5 readtype.p: type-name inserted into symbol table before processing
  3.6 symbol.p: parameters distinguished; no output of standard names
4 Revalidation of the software
  4.1 Integration testing
  4.2 Regression testing
    4.2.1 Test Cases
      4.2.1.1 Test Results
  4.3 Documentation of changes made to 'pxr' source code
  4.4 Configuration management and Version Control
5 Executive Summary
```

Figure 10.16: The Maintenance History of pxr

10.2.5 Future maintenance of 'pxr'

In the construction of the book to produce version 1.1 of 'pxr', the editing operations assumed greater importance than the operations associated with navigation and retrieval of information. In this part of the maintenance assignment the reverse is true, as described below.

Future maintainers of 'pxr' may need to acquire an understanding of the current maintenance assignment in order to perform their maintenance assignment. Browsing is one way of achieving such an understanding, using a *conceptual map*. The conceptual map of the book documenting the maintenance of 'pxr' has been shown as an indented table of contents so reflecting its hierarchical structure, which helps to orient the user when browsing.

This conceptual map is itself an *abstraction*, a well-known example of such an abstraction is a road map, with towns as nodes in the directed graph. One of the problems associated with navigation within a large data structure is disorientation. An obvious benefit of a hierarchical topology is in combatting this disorientation, the operation which most obviously characterises the book structure as a conceptual *map*, is **path** which prints out the node keys from the root of the tree, to a nominated node, thus giving the *co-ordinates* of the node, orienting the user in his/her perception of their position on the conceptual map, and providing valuable traceability information.

As an example of the use of **path**, the path from the title of the book to the node 2.2.1 gettoken.p is shown by Figure 10.17 below:

The **path** operation places the module paramlist.p in its rightful context, orienting the

```
Title  Version 1.1 pxr
2 Understanding the software
2.2 Local Understanding
2.2.2 paramlist.p: parsing  formal and actual parameters.
```

Figure 10.17: Output from the 'path' operation

maintenance programmer.

To further understanding, the **evaluate** operation can be used to display the contents of node. During a browsing session many nodes may be visited, the collection of nodes constituting a 'virtual' book. Sometimes ideas are half-formed and then the tenuous thread which holds the idea together breaks, and the idea is lost. The **evaluate** operation also keeps a record of which nodes have been visited during a browsing session, storing the nodes in a list, and this list can be displayed using the **trail** operation. By this means it may be possible to recapture the idea which was lost.

An additional means of aiding understanding is provided by the **abstract** operation which is a 'slicing' operation offering a *view* of the documentation. **Abstract** produces a list of nodes concerned with the same topic, the nodes being the components of a hierarchy, e.g. sections, subsections, etc. Information is made available at greater levels of detail by the hierarchical decomposition, and can be utilised in the same way as texts in programmed learning. In Figure 10.18 below the operation is used to provide information concerning the revalidation of the software.

```
4 Revalidation of the software
4.1 Regression testing
4.1.1 Test Cases
4.1.1.1 Test Results
```

Figure 10.18: Output from the 'abstract' operation

Support for traceability

An identifier, e.g. 1.2.1.1_XYZ_0 provides a path which can be traced backwards through the tree of called components, to show where the component was called from. The call tree could also be displayed in graphical form.

10.2.6 Other attributes of the documentation paradigm

Relational aspect

Storing the information concerning source code as a hierarchy of tables with one of the nodes containing the information produced by, e.g. program analysis tools, means that queries of a relational nature can also be supported. The relational model offers more flexibility than other data models, an important consideration since there is no basis for determining in advance which type of questions will be most frequently asked, by maintainers.

10.2.7 Weaknesses associated with the documentation paradigm

The main disadvantage of hierarchical information structures is concerned with their update, which is difficult when the node to be inserted or removed is not a leaf node, depending on the granularity of the object.

There is no automatic provision of facilities for versioning of understanding. It is to be expected that understanding of a program is achieved in an incremental fashion, and an interface would need to be provided to a tool such as RCS [123].

The documentation paradigm does not support a rule-based query language, as it lacks a sophisticated database management system.

10.3 The effect of incomplete use of the toolset by maintainers

The incomplete use of the toolset will degrade its effectiveness. It is not possible to quantify this, but the consequences of its incomplete use are set out below.

During the use of the documentation paradigm there are various activities carried out by the members of the maintenance organisation. Some of these activities depend on one particular tool. If the toolset is used incompletely, then it is not possible to say quantitatively what effect this would have on the maintenance activity, unless it is also known what approach

is to be used in tandem with the documentation paradigm, in lieu of relying solely on the toolset.

The toolset makes it possible to express the documentation paradigm; failure to use certain tools may render invalid any attempt to use other tools.

10.3.1 Reasons for using the complete toolset

The operations which comprise the paradigm are a *minimal* set, and the importance of this fact is described below.

1. **Process Structuring**

   The documentation paradigm acts as the conceptual schema for the ISMSE, and encapsulates the process model underlying it: i.e. the anthology's structure reflects the conceptual schema for the ISMSE. Evolution of the process model underlying the ISMSE requires that the ISMSE *must be able to support its own evolution.* It is therefore vital to ensure that the environment is always able to support the evolution of this process model through the activity of process-structuring, utilising the indispensable operations which make it possible to build and edit the anthology structure. Without the tools to build and edit the process model underlying the conceptual schema, the evolution of the process model would not be possible.

2. **Effect on the technical aspect of maintenance**

Some of the operations have a one-to-many mapping to the activities performed during the maintenance of software, as can be seen from the table below in table 10.1. Removal of a particular operation from the set may have a deleterious effect on the effective performance of these activities, or may render them impossible to perform. The operations that could possibly be dispensed with are those which make it possible to extract information from the book structure, for the purpose of understanding. It is possible to mimic these operations manually, but this would mean that these tasks are very time-consuming, and since understanding is the rate-determining step in the maintenance activity, this approach would be counter-productive. In this context perhaps the most crucial operation is the Cross-reference operation, since it has the greatest potential to aid in understanding the software system being maintained.

3. **Management**

The operations which extract information from the book structure are vital to the management function - without them it is difficult to bring maintenance under management control. Management is closely linked to organising, monitoring and auditing, and without the use of the Table, Abstract and Evaluate operations, it is impossible to monitor the progress of the maintenance assignment. In addition, the absence of the operation Cross-reference makes it impossible to extract information concerning the maintenance assignment, for the purposes of auditing. The absence of the Cross-reference operation makes impossible the analysis of the change requests with a view to batching some of them, using the scratchpad facility; batching being an activity concerned with organising.

4. **Desirable characteristics of the ISMSE lost as a result of incomplete use of the toolset**

A summary of the activities supported by the complete toolset is shown below in Table 10.1. Perhaps the best way of describing the effect of the incomplete use of the toolset is to look at the desirable characteristics conferred upon the ISMSE by the toolset - and how some of these properties would be 'lost', if certain operations were not used, making some activities impossible.

Activity	Operations involved
Documenting	Build and Edit
Browsing	Cross-reference, Trail
Searching	Cross-reference, Trail
Understanding	Cross-reference, Trail
Authoring	Build and Edit
Editing	Build and Edit
Versioning	Build and Edit, Equal
Tracing	Cross-reference, Trail
Auditing	Cross-reference, Trail
Managing	Table, Cross-reference, Abstract, Evaluate
Process Structuring	Build and Edit

Table 10.1: Operations and activities associated with the documentation paradigm

Abstraction - directly supported by an operation of the same name.

Extensibility - support for configuration management and versioning, using, e.g. the Equal operation.

Grouping of Resources - pooling of efforts which makes use of the Cross-reference operation, and Build and Edit operations.

Adaptability - achieved through incremental implementation, making possible incremental integration, to reflect evolution of the process model. The operations involved are Build and

Edit operations and those operations which facilitate understanding.

Tailorability - integrating maintenance process and maintenance organisation. Evolution of the framework which makes integration possible requires, at least, the use of Build and Edit operations.

Unification - facilitates communication and co-ordination between component parts of maintenance organisation, requiring operations to support monitoring and auditing.

10.4 The scope for reuse of experience within the proposed ISMSE

The ensuing knowledge stemming from the information collected during a maintenance assignment can provide a record of the experience gained during the assignment, which can be of use in future maintenance assignments. In time the maintenance-history becomes a resource, containing much information concerning maintenance strategies, providing a tool for the advancement of knowledge concerning software maintenance, and serving to advance software maintenance research. An example of this is in the area of re-use; previous assignments can be studied to see how particular problems were solved, which can aid in achieving an increase in productivity. A failure to learn from the past often means that the mistakes of the past are destined to be repeated, which will *reduce* the productivity of the organisation. The establishment of reusable processes is the most effective way of providing for reuse in an environment. Mapping the chapter concerning the change request to the succeeding chapters

which execute this change request provides the reader with a means of re-using parts of the maintenance process.

10.5 How managers could incorporate 'milestones'

The software development life-cycle is separated into phases for the purpose of incorporating 'milestones' for managers associated with the project. Similarly the maintenance of software is similarly partitioned, hierarchically, using phases. The maintenance model maps directly onto the book structure and the degree of completion of each chapter is an indicator of the progress of the maintenance assignment. Apart from the first phase, no phase of maintenance can begin until the previous phase is completed, and so the table of contents operation provides managers with a means of monitoring the progress of the assignment.

10.6 The scope for using the ISMSE to document its own development

The documentation paradigm in this thesis enables the documentation of the maintenance process, which is a hierarchical one. Any process having a hierarchical nature is capable of being documented using the documentation paradigm promulgated in this thesis, since

the book structure is itself a hierarchical one. Furthermore, maintenance is a microcosm of software development and it has been demonstrated that it can be documented using the documentation paradigm. By induction, the software development process is also capable of being documented using the same paradigm. If, say, an IPSE was used to build an ISMSE, the documentation paradigm could be hosted on the IPSE. The only other alternative is to record the development on paper, and then at some later stage make the transition to the electronic version of the documentation paradigm.

10.7 Summary

The use of the documentation paradigm in a toolkit environment, has been described, using the host computer's hierarchical file structure to mirror the structure of the ADT Maintenance_History, and operating system commands to mirror the operations which define the Maintenance_History as an ADT. The book structure is more ordered than a normal acyclic directed graph, because a relationship exists between the nodes, the relationship being stronger than the simple inequality relation existing between nodes in many binary search trees. Furthermore, the operations which characterise the documentation paradigm enable the maintainer to abstract away from the underlying file representation, and view the information as collections of entities, or objects. In this way the ADT Maintenance_History is mapped to the underlying file system, the minimum granularity of the operand for its operations being the file, the relationships between files which are *logically* related are maintained. The editing functions which help define the ADT Maintenance_History, help to underline

the fact that the documentation paradigm provides the basis for an authoring system, which is essential to achieve the desired level of productivity when implementing computer-aided learning (CAL), one of the functions of the ISMSE. CAL applies to understanding the software system being maintained, and to learning to use a complex tool, i.e. the ISMSE itself. The documentation paradigm provides a means of integrating these two functions of the ISMSE, within a single data structure.

The Maintenance History provides a log of maintenance activities, and the inclusion in the maintenance log of explanatory sections detailing *why* something was done in the way it was provides an insight into the strategy adopted during software maintenance and can aid the understanding of the lower-level activities, including the coding phase, performed by 'current' maintainers, by *future* maintainers.

The documentation paradigm confers several of the desirable characteristics enumerated by Magel [77] upon the ISMSE. It has been shown that the paradigm supports **abstraction**, **extensibility**, and **grouping of resources**. The documentation paradigm is independent of any programming language or process model, and confers **generality** upon the ISMSE. Evolution of the process model will not invalidate the use of the paradigm; since the paradigm *is* the conceptual schema for the ISMSE, and supports **incremental implementation**, through incremental integration, and therefore confers **adaptability** upon the ISMSE. Moreover the documentation paradigm fosters a disciplined approach to the documentation of software, and provides the framework for integrating the maintenance process and the maintenance organisation into a single unifying structure, so the paradigm confers **tailorability** upon the ISMSE. This also simplifies communication and coordination

between the component parts of the organisation, particularly the technical members and management members of the maintenance organisation, so conferring **unification** upon the ISMSE.

The application of the documentation paradigm to the maintenance of a software tool, has been used to assist in its evaluation and to extrapolate its use to a larger software system. In particular, the analysis of the paradigm highlighted the support it offers for abstracting, documenting, browsing, searching, understanding, authoring, editing, versioning, tracing, auditing, managing, and process structuring.

From this evaluation of the documentation paradigm, it can be inferred that it would provide adequate functionality, when used to document the maintenance of a large software system, particularly as regards safeguarding the completeness and consistency of the system's documentation; and supporting configuration management, version management, and project management.

Chapter 11

Conclusions and Further Work

11.1 Review of the work

The work in this thesis is reviewed here and suggestions made as to how the work can be extended in the future.

The original objectives of this thesis were to provide a maintenance organisation with the means of reducing the maintenance backlog and narrowing the 'hardware–software gap'. A means of increasing productivity, by the use of an Integrated Software Maintenance Support

Environment, has been outlined, using a documentation paradigm within the framework of this environment, to provide an effective strategy for the maintenance of software. Specifically the thesis aimed to:

1. examine the need for a maintenance support environment and the need for a strategy for software maintenance
2. investigate currently-available support environments for their support for software maintenance
3. develop a strategy for the maintenance process, and a high level design for a maintenance support environment
4. formally define and implement a maintenance strategy and demonstrate and analyse its use

11.2 Have the objectives been achieved

The ever-increasing complexity of software systems, and the size and complexity of the their associated documentation, revealed the need for an integrated support environment for software maintenance. This need defined the main objective of the research, that of devising a strategy for providing automated support for software maintenance, particularly with regard to the use of software tools to gather information concerning the source code.

and its associated external documentation, and a means of recording the documentation of maintenance performed on the system.

A basis for a disciplined approach to providing automated support software maintenance is an underlying process model. Devising such a model refined the main objective of the research. The literature was surveyed to see how existing software maintenance process models served as a means of providing the basis for a disciplined approach, paying particular attention to those models which focused on the role of understanding in the maintenance of software. It was found that the main deficiency in existing maintenance process models for software maintenance was their restriction to the technical aspects of software maintenance. The process model devised for software maintenance in this thesis acknowledges the importance of the maintenance organisation, particularly with regard to its information requirements, and the bearing this has on the planning and monitoring of software maintenance. These particular aspects of software maintenance had an important bearing on the design of the ISMSE, and on the formulation of a strategy for software maintenance, resulting in the documentation paradigm, adopted to support this strategy.

Having devised a process model for the maintenance of software it was then necessary to find an environment which could host this model, so that automation could play its part in increasing the productivity of a maintenance organisation. An overview of integrated software engineering support environments was undertaken to examine those characteristics of the environments which provide support for a disciplined approach to software maintenance.

This literature survey revealed that it is the architecture and interfaces which are the dom-

inant characteristics of integrated software engineering support environments, since the architecture is the implementation of the environment's design and the interface plays a vital part in the integration of the environment's functions.

Existing integrated software engineering support environments were surveyed, those that were commercially-available, and also academic research environments. It was found that most of these environments were not truly integrated, true integration was found to exist only in Integrated Project Support Environments (IPSEs). These environments aim to support the complete software development life cycle, but most of their support is aimed at software development, since they do not provide support for the understanding of software, their toolset lacking the necessary tools to support analysis of source code. The literature survey revealed a need for an Integrated Software Maintenance Support Environment (ISMSE), together with the need for a disciplined engineering approach to software maintenance.

Having established the need for an ISMSE, and the implicit link between the software maintenance process model and a maintenance organisation, the next objective was to propose a high-level design for the ISMSE, based largely on the information requirements of a maintenance organisation. Within the framework of a high-level design the conceptual schema for an environment database has been devised to provide a documentation paradigm to support a strategy for the maintenance of software. This strategy aims to make the *future* understanding of software easier, while at the same time safeguarding the consistency and completeness of its documentation; a vital requirement since the understanding of source code is most easily achieved from its high-level external documentation.

A high-level design for an ISMSE has been proposed, based largely on the information requirements of a maintenance organisation, and a study of the mechanisms necessary to capture information, and provide reliable documentation of changes to source code. Within this framework a conceptual schema for an environment database has been devised to provide a documentation paradigm to support a strategy for the maintenance of software. This strategy aims to make the future understanding of software easier, while at the same time safeguarding the consistency and completeness of its documentation, a vital requirement since the understanding of source code is most easily achieved from its high-level documentation.

A study of the mechanisms necessary to capture information, and provide reliable documentation of changes to source code made it possible to establish the information requirements of a maintenance organisation, since an environment is primarily concerned with information, and in particular the link between information and knowledge. Knowledge implies understanding, and achieving this understanding is the 'rate determining step' in the maintenance of software. Factors which aid knowledge-capture are the easy storage, retrieval and processing of information. This led to the conclusion that the database in the ISMSE has a vital role to play in the understanding of software, providing an additional impetus to devising a conceptual schema for the database, so that the information in the database could be structured, providing support for abstraction to aid understanding.

The conceptual schema for the environment database, which is central to the documentation paradigm, has been formally defined as an abstract data type, and the evaluation of the prototype has confirmed its utility and efficacy, ignoring performance considerations.

Summary

The primary objectives of the research have been achieved, but in the long term, in the wider sphere of support environments, much work remains to be done to make the ISMSE a viable system for the maintenance of large software systems; the next section describes some of the further work required to achieve this aim.

11.3 Further Work

Central to the working of the ISMSE is its integration mechanism, which unifies the functions of the environment so that it functions as a single tool. In addition, the ISMSE needs to be made active, suggesting approaches and tools to the maintainer, for the maintenance of software, rather than simply functioning passively; this could be achieved by making the change-request machine-interpretable. Thus, it could then be ascertained whether the change request has previously been satisfied, avoiding the needless repetition of work, and secondly, the change request can also serve as a template for the structure of the book, defining the contents of the chapters within which the maintenance organisation will conduct the maintenance assignment. This approach has great utility if the template is capable of parameterisation, for a particular type of maintenance, e.g. corrective maintenance, providing a standardised approach to the maintenance of software, leaving the maintenance team to concentrate on the content of the book, without having to be concerned with its structure, avoiding a 'cognitive overhead'. Parameterisation of the template also offers support for user-enhanceable systems, so that tailoring of the documentation paradigm is possible. In

support of the above, the design of a suitable object management system (OMS) for the ISMSE, is paramount, for the following reasons:

1. An OMS reflects the complexity and granularity of the objects dealt with during the course of a maintenance assignment, at a higher semantic level than classical file systems or DBMS, through the enforcement of constraints, using ADTs. This has implications for the understanding phase of maintenance, which is the rate-determining step in a maintenance assignment. Moreover, an OMS offers better services for the storage and retrieval of these complex data structures than classical file systems or DBMS.

2. An interface between the OMS and software tools is required, so that information can be entered into the book structure, without manual intervention by the maintainer, if desired. This requires the automatic invocation of one tool by another tool, to extract information concerning the software: i.e. the OMS facilitates tool-tool communication, and so its design must include an abstract interface so that evolution of the conceptual schema can proceed, reflecting the evolution of the maintenance process, without the need to alter the ISMSE's toolset.

3. Collaborative authoring by members of the maintenance team must be supported, which, in turn, means that a view mechanism must be devised, so that different categories of user can access the data structure simultaneously, without the need for complex locking mechanisms, often an integral part of commercial DBMS, and unacceptable in this context, because of the long time-spans involved in many transactions.

As pointed out in chapter 10, the technology underlying ISEs is immature and so in the short term the documentation paradigm must be implemented using available technology, using, e.g. an available software engineering environment, so that the documentation paradigm can be mapped onto an underlying hierarchical file structure, the operating system's tools being used to implement some of the operations which underlie the documentation paradigm. The first priority is the design of a suitable implementation of the documentation paradigm using an appropriate high-level language, which is able to interface with the operating system, for the purpose of creating and editing a directory structure, and also able to interface with a DBMS which is compatible with a hierarchical file structure. A selection of appropriate commercially-available software tools for maintenance then needs to be integrated with the prototype ISMSE, and interfaced with the DBMS. The set of operations underlying the documentation paradigm could be expanded to offer the facilities of a sophisticated authoring system.

In the area of re-use an indexing system needs to be devised so that reusable processes, e.g. designs, algorithms, specifications, code fragments, can easily be found.

Bibliography

[1] Akscyn R.M. et al, **'KMS: A distributed hypermedia system for managing knowledge in organisations'**, *Comms. ACM. Vol. 31 No. 7, July 1988*

[2] Alderson A. et al. **'An overview of the Eclipse Project'**, in *Integrated Project Support Environments, ed. J. McDermid, Peter Peregrinus Ltd., 1985*

[3] Alford M., **'A Requirements Engineering Methodology for Real-Time Processing Requirements'**, *Trans. Software Eng., Vol. SE-3, No. 1, January 1977*

[4] Ambras J., O'Day V., **'Microscope: A program analysis system'**, *Proc. 20th Annual Hawaii International Conference on System Sciences, 1987*

[5] An K.H., Gustafson D.A., Melton A.C, **'A model for software maintenance'**, *Proc. IEEE Conference on Software Maintenance 1987, Austin, Texas*

[6] Arthur L.J., **'Improving Software Quality'**, Wiley, 1993

[7] Balser R., Cheatham T.E., Green C., **'Software Technology in the 1990s : Using a New Paradigm'**, *IEEE Computer, November 1983, p39-45*

[8] Belady L.A., Lehman M.M., **'A model of large program development'**, *IBM System Journal, Vol. 15, No. 3, 1976*

[9] Bennett K.H., **'The Software Maintenance of Large Software Systems: Management, Methods and Tools'**, in *Software Engineering for Large Software Systems, ed. B.A. Kitchenaum, Elsevier Science Publishers Ltd., 1990*

[10] Benington H.D., **'Production of Large Computer Programs'**, *Proc. ONR Symposium on Advanced Programming Methods for Digital Computers, June 1956, p350-361*

[11] Bigelow J., **'Manipulating Source Code in DynamicDesign'**, *Hypertext '87, IEEE 1987*

[12] Bigelow J. **'Hypertext and CASE'**, *IEEE Software, March 1988*

[13] Biggerstaff T., Ellis C., Halasz F., Kellog C., Richter C., Webster D. **'Information Management Challenges in the Software Design Process'**, *Technical Report STP-039-87, MCC, Software Technology Program, January 1987*

[14] Boehm B.W., **'The high cost of software'**, *Proc. Symp. on High Cost of Software, Monterey, California, 1973*

[15] Boehm B.W., **'Software Engineering'**, *IEEE Transactions on Computers Vol.C-25, No.12 December 1976*

[16] Boehm B.W., Brown J.R., Lipow M. **'Quantitative evaluation of software quality'**, *Proc IEEE/ACM Second Int. Conf. Software Eng., October 1976*

[17] Boehm B.W., **'Software Engineering Economics'**, Prentice Hall 1981.

[18] Boehm B.W., **'A Spiral Model of Software Development and Enhancement'**, *ACM Sigsoft Software Engineering Notes, Vol. 11, No. 4, August 1986*

[19] Borgida A., et al, **'Knowledge Representation as a Basis for Requirements Specification'**, *IEEE Computer, Vol. 18, No. 4, 1985*

[20] Brooks R., **'Towards a Theory of the Comprehension of Computer Programs'**, *Int. J. Man-Machine Studies, Vol. 18, 543-554, 1983*

[21] Brooks F.P., **'Essence and accidents of software engineering'**, *IEEE Computer, April 1987*

[22] Bush V., **'As we may think'**, *Atlantic Monthly, July 1945*

[23] Buxton N., **'Requirements for Ada Programming Support Environments'**, *Stoneman, DOD, February 1980*

[24] Carter G.W., **'Seven Stages of Maintenance'**, *Software Maintenance News, p14, 1986*

[25] Chapin N., **'Software Maintenance with Fourth Generation Languages'** *ACM Sigsoft Software Engineering Notes, Vol. 9, No. 1, January 1984*

[26] Church A., Rosser J., **'Some properties of Conversion'**, *Trans Amer. Math. Soc. 39, 472-482, 1936*

[27] Clemm G.M., **'The Odin Environment Integration Mechanism'**, *Technical Report CU-CS-323-86, University of Colorado, Boulder, Colorado, April 1986*

[28] Clocksin W.F., Mellish C.S., **'Programming in Prolog'**, Pub. Springer-Verlag 1981

[29] Collofello J., Orn M., **'A Practical Software Maintenance Environment'**, *Proc. IEEE Conference on Software Maintenance, 1988*

[30] Conklin J., **'Hypertext: An Introduction and Survey'**, *IEEE Computer, September 1987*

[31] Dart S., Carnegie Mellon University, **'Software Development Environments'**, *IEEE Computer, November 1987*

[32] Delisle N.M., Menicosy D.E., Schwartz M.D. **'Viewing a programming environment as a single tool'**, *ACM Sigplan Notices, May 1984, Vol. 19, No. 5*

[33] Delisle N.M., Schwartz M., **'Neptune: A hypertext system for CAD applications'**, *Proceedings ACM Sigmod '86, New York*

[34] DeMarco T., **'Structured Specification and Systems Analysis'**, Yourdon Press 1981

[35] Distaso J.R., **'Software Management - A Survey of the Practice 1980'**, *IEEE Proceedings, September 1980, p1103-1119*

[36] Donahoo J. D., Swearinger D., Rome Air Development Centre, Grifiss AFB, New York, **'A review of Software Maintenance Technology'**, *RADC-TR-80-13, February 1980*

[37] Fairley R., **'Software engineering concepts'**, Pub. McGraw-Hill, 1985

[38] Feldman J., **'Make: A Program for Maintaining Computer Programs'**, *Software Practice and Experience, April 1979*

[39] Fjeldstad R.K., Hamlen W.T., **'Application program maintenance study'**, in *Tutorial on Software Maintenance. Silver Spring, MD: IEEE Computer Society Press, 1983*

[40] Foster J.R., Jolly A.E.P., Norris M.T., **'An overview of software maintenance'**, *Br Telecom Technol J. Vol. 7, No. 4, October 1989*

[41] Freedman D.P., Weinberg G.M., **'A checklist for potential side-effects of a maintenance change'**, in *Techniques of Program and System Maintenance, ed. Girish Parikh, Ethotech,. Inc., 1980, pp 61-68*

[42] Garg P.K., Scacchi W., **'On Designing Intelligent Hypertext Systems for Information Management in Software Engineering'**, *Hypertext '87 TR88-013, University of North Carolina, November 1987*

[43] Garg P.K., **' Abstraction mechanisms in Hypertext'**, *Commcns. ACM, Vol. 31, No. 7, July 1988*

[44] Giddings R.V., **'Accommodating Uncertainty in Software Design'**, *Comms. ACM, May 1984, p428-434*

[45] Glagowski T.G., **'Using a relational query language as a software maintenance tool'**, *Proc IEEE Compsac 1985*

[46] Glass R.L., Noiseaux R.A., **'Software Maintenance Guidebook'**, Pub. Prentice-Hall, 1981

[47] Goguen J.A. et al. '**Abstract Data Types as Initial Algebras and Correctness of Data Representations**'. *Proc. Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975*

[48] U.S.A. GSA. '**A Software Tools Project: A means of capturing technology and improving engineering**'. *Report OSD-82-101, Office of Software Development, and Information Technology, Federal Software Testing Centre, 1982*

[49] U.S.A. GSA. *Report OSD-82-101, Office of Software Development, and Information Technology, Federal Software Testing Centre, 1984*

[50] Guttag J., Horowitz E., Musser D.R., '**Abstract Data Types and software validation**'. *USC Information Sciences Institute Technical Report, ISI/RR-76-48, 1976*

[51] Guttag J., '**Abstract Data Types and the Development of Data Structures**', *Comms. ACM, Vol. 20, No.6, June 1977*

[52] Hall J.A. et al. '**An overview of the Aspect Architecture**' in *Integrated Project Support Environments*, ed. J. McDermid, Peter Peregrinus Ltd., 1985

[53] Hansen G., '**Who says there have been no advances in Software Maintenance Tools ?**' in *Software Maintenance News, Vol. 5, No. 12, p6, December 1987*

[54] Henderson P.B., *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments - ACM, Sigplan Notices, Vol. 22, No. 1 Palo Alto, California, December 9-11, 1986*

[55] Houghton R.C., 'Characteristics and Functions of Software Engineering Environments: An Overview' *ACM Sigsoft Software Engineering Notes, Vol. 12 No. 1, January 1987*

[56] Hutchison D., Walpole J., **'Eclipse - a distributed software development environment'**, *IEE Software Engineering Journal, March 1986*

[57] **'IEEE Standard Glossary of Software Engineering Terminology'**, *IEEE 1983*

[58] **'IEEE Guide to Software Requirements Specifications'**, *IEEE 1984*

[59] Jones C.B., **'Systematic Software Development Using VDM'**, Pub. Prentice-Hall, 1986

[60] Kaiser G., Feiler P., **'Intelligent assistance without artificial intelligence'**, *Proc 32nd IEEE Computer Society International Conference on Software Engineering, February 1987*

[61] Kaiser G. et al, **'Intelligent assistance for software development and maintenance'**, *IEEE Software, May 1988*

[62] Kempe M., **'Hyperbook: an experiment with PCTE'**, *ACM Sigsoft Software Engineering Notes, Vol. 14, No. 5, July 1989*

[63] Koenig S., **'ISEF - An Industrial Strength Software Engineering Framework'**, *ACM Sigsoft '88: 3rd Symposium on Software Development Environments (SDE3)*

[64] Lehman M.M., **'A Further Model of Coherent Programming Processes'**, *Proceedings of Software Process Workshop, IEEE, February 1984, p27-33*

[65] Lehman M.M., Turski W.M., **'Essential properties of IPSEs'**, *ACM Sigsoft software engineering notes, Vol. 12 No. 1, January 1987*

[66] Letovsky S., **'Cognitive Processes in Program Comprehension'**, *Proceedings of the Conference on Empirical Studies of Programmers 1986*

[67] Letovsky S. and Soloway E., **'Delocalised plans and program comprehension'**, *IEEE Software, May 1986*

[68] Leung H.K.N., White L., **'Insights into Regression Testing'**, *Proc. IEEE Conf. on Software Maintenance, Miami, 1989*

[69] Lewerentz C., **'Extended Programming in the large in a software engineering environment'**, *ACM Software Engineering Notes, Vol. 13, No. 5, November 1988*

[70] Lientz E.B., Swanson K., **'Characteristics of Application Software Maintenance'**, *Commcns. ACM Vol. 12, p466-471, June 1978*

[71] Linger R.C., Mills H.D., Witt B.I., **'Structured Programming: Theory and Practice'**, Pub. Addison-Wesley 1979

[72] Linger R.C., **'Software Maintenance as an Engineering Discipline'**, *Proc. IEEE Conference on Software Maintenance 1988*

[73] Linton M.A., **'Implementing relational views of programs'**, *ACM SE Notes, Vol. 9, No. 3, May 1984*

[74] Liskov B.H., Berzins V., **'An appraisal of program specifications'** in *Research Directions in Software Technology, ed. P. Wegner 1979, Cambridge, Mass. MIT Press.*,

[75] Littman D. et al. '**Mental models and software maintenance**'. *The Journal of Systems and Software. Vol. 7. 1987*

[76] Liu C., '**A look at software maintenance**', *Datamation. November 1976*

[77] Magel K., '**Principles for Software Environments**', *ACM Software Engineering Notes. Vol. 9. No. 1. January 1984. 32*

[78] Mair P., '**Integrated Project Support Environments - State of the Art Report**', *NCC 1986*

[79] Martin J., McClure C., '**Software maintenance, the problem and its solutions**'. Prentice Hall. London 1983

[80] McCracken D.D., Jackson M.A., '**Life-Cycle Concept Considered Harmful**', *ACM Software Engineering Notes. April 1982. p29-32*

[81] McGuffin R.W., Elliston A.E., Tranter B.R. and Westmacott P.M. '**CADES - Software Engineering in Practice**'. *Proc. 4th Int. Conf. on Software Engineering, Munich 1979*

[82] McKissick J.M., Price R.A., '**The Software Development Notebook**', *Proc. 1979 IEEE Annual Reliability and Maintainability Symposium*

[83] Mellor P., '**Field Monitoring of Software Maintenance**', *Software Engineering Journal, January 1986*

[84] Meyer B., '**On Formalism in Specifications**'. *IEEE Software. January 1985*

[85] Mills H.D., **'Top-Down Programming in Large Systems'** in *Debugging Techniques in Large Systems*, R. Rusking (ed), Prentice-Hall, 1971 p41-55

[86] Narayanaswamy K. and Scacchi W. **'An environment for the development and maintenance of large software systems'**, in *Proc 2nd SOFTFAIR, IEEE Comput. Soc., 1985*

[87] Narayanaswamy K., **'A framework to support software system evolution'**, *Ph.D. Dissertation, Univ. Southern California, May 1985*

[88] Narayanaswamy K., Scacchi W. **'A database foundation to support software system evolution'**, *J.Syst. Software, 1987*

[89] Narayanaswamy K., Scacchi W., **'Maintaining Configurations of Evolving Systems'**, *IEEE Trans. Soft. Eng., Vol. SE-13, No. 3, March 1987*

[90] NBS. **'Features of software development tools, special publication'**, *500-74, U.S. National Bureau of Standards, 1980*

[91] Nelson T.H., **'Getting it out of our system'**, *Information Retrieval, A Critical Review, G. Schecter Ed., Thompson Books, Washington D.C., 1967*

[92] Notkin D., **'The Gandalf Project'**, *Journal of Systems and Software, Vol. 5, No. 2, May 1985*

[93] Notkin D., **'The Relationship Between Software Development Environments and the Software Process'**, *ACM Sigsoft '88 3rd Symposium on Software Development Environments, Boston Mass., 1988*

[94] Oman P. W., Cook C. R., **'The book paradigm for improved software maintenance'**, *IEEE Software, Jan 1990*

[95] Oren T., **'The architecture of Static Hypertexts'**, *Hypertext '87, TR88-013, University of North Carolina, March 1988*

[96] Osborne W.M., Martin R.J., **'Guidance of Software Maintenance'**, *Nat. Bureau of Standards, NBS Special Publication, 500-106, December 1983*

[97] Osterweil L.J., **'Software Environment Research Directions for the next Five Years'**, *Computer 14, p 35-43, April 1981*

[98] Osterweil J. et al. **'ODIN: An integration mechanism for an software engineering environment'**, *University of Colorado Technical Report, 1989*

[99] Osterweil L., **'Software processes are software too'**, *Proc 9th Int. Conf. Soft. Eng. Monterey, March 1987*

[100] Parikh G., **'Some tips, techniques, and guidelines for program and system maintenance'**, in *Techniques of program and system maintenance, Winthrop Publishers, Cambridge MA., 1982, 65-70*

[101] Parnas D.L., **'Designing Software for Ease of Extension and Contraction'**, *IEEE Trans. Soft. Eng., March 1979, p128-137*

[102] Patkau B.H., **'A foundation for software maintenance'**, *M.Sc. Thesis, Department of Computer Science, University of Toronto, December 1983*

[103] Penedo M., **'Prototyping a Project Master Database for Software Engineering Environments'**, *ACM Sigplan Notices, Vol. 22 No. 1 Jan 1987*

[104] Petzold K., 'The COBOL maintenance crisis'. *First Software Maintenance Workshop, University of Durham, England, September 1987*

[105] Pressman R.S., **'Software Engineering - A Practitioner's Approach'**. Third Edition. Pub. McGraw-Hill, 1992

[106] Ramamoorthy C.V., **'Genesis - an integrated environment for supporting development and evolution of software'**. *Proc. IEEE Compsac 1985*

[107] Ramamoorthy C.V., Usada Y., Tsai W., Prakash A., **'Genesis: An Integrated Environment for Supporting Development and Evolution of Software '**, *Proc. IEEE Compsac. 1985*

[108] Raskin J., **'The hype in hypertext: A critique'**. *Hypertext '87. TR88-013. University of North Carolina. March 1988*

[109] Riddle W.E. et al. **'The stars program - overview and rationale'**. *IEEE Computer. November 1983*

[110] Riddle W.E., **'The Evolutionary Approach to Building the Joseph Software Development Environment'**. *Proc IEEE Softfair - Software Development Tools, Techniques and Alternatives, p 317-325, 1983*

[111] Riddle W.E., **'Improving the Software Process'**. *Proc 9th Int. Conf. Soft. Eng. Monterey, March 1987*

[112] Ross D., **'Structured Analysis (SA): A Language for Communicating Ideas'**, *Trans. Software Eng., Vol. SE-3, No. 1, January 1977*

[113] Royce W.W., 'Managing the Development of Large Software Systems:Concepts and Techniques', *Proceedings, WESCON, August 1970*

[114] Schneiderman B., Mayer R., **'Syntactic/Semantic Interactions in Programming Behaviour: A Model'**, *Int. J. Computer and Information Science, Vol. 8, No. 3, 1979*

[115] Selig F., **'Documentation Standards'**, in *Software Engineering, Proc. of meeting, Garmisch, Germany, October 1968*

[116] Sharpley W.K., **'Software maintenance planning for embedded computer systems'**, *Proc. IEEE Compsac 77, November 1977*

[117] Snowdon R.A., **'CADES and software system development'**, *Software Engineering Environments, ed Hunke, H., page 81-96, North-Holland June 1980*

[118] Sommerville I., **'Software Engineering'**, Third Edition, Pub: Addison Wesley 1989

[119] Spivey J.M., **'Introducing Z: A specification language and its formal semantics'**, Pub. Cambridge University Press, 1988

[120] Stenning V., **'On the role of an environment'**, *Proceedings of the 9th International Conference on Software Engineering, ACM, 1987*

[121] Swanson E.B., **'The dimensions of maintenance'**, *Proc IEEE/ACM Second Int. Conf. Software Eng., October 1976*

[122] Teichrow D., Hershey, E., **'PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems'**, *Trans. Software Eng., Vol. SE-3, No. 1, January 1977*

[123] Tichy W., **'Design, Implementation, and Evaluation of a Revision Control System'**, *Proc. 6th ICSE, Tokyo, Japan, 1982*

[124] Tsichritzis D.C., Klug A., **'The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems'**, *Information Systems, Vol. 3, pp 173-191, 1978*

[125] Tully C.J., **'Prospects for Future Environments'**, *Proc. 9th Int. Conf. Soft. Eng. Monterey, March 1987*

[126] Weidermann N.H., Habermann A.N., Borger M.W., Klien M.H., **'A methodology for evaluating Environments'**, *ACM Sigplan Notices, Vol. 22A No. 1 , December 1986*

[127] Weiser M., **'Programmers use Slices when Debugging'**, *Comms. of the ACM, Vol. 25, No. 7 July 1982*

[128] Wing J., **'A Specifier's Introduction to Formal Methods'**, *IEEE Computer, September 1990*

[129] Yau S., Collofello J.S. and McGregor T., **'Ripple Effect Analysis of Software Maintenance'**, *Proc. IEEE COMPSAC 78, Chicago, IL., November 1978, 60-65*

[130] Yau S.S. et al, **'A methodology for software maintenance'**, in *Proc. Intern. Comput. Symp., 447-458, December 1982*

[131] Yeh R.T., Zave P., **'Specifying Software Requirements'**, *Proc. IEEE, Vol. 68, No. 9 September 1980*

[132] Zahniser R.A., **'The perils of top-down design'**, *ACM SE notes, Vol. 13, No. 2, April 1988*

[133] Zilles S.N., **'Abstract Specifications for Data Types'**, *IBM Res. Lab., San Jose, California, 1975*