Requirements for a software maintenance support environment

Simon, Amaury

University of Durham

School of Engineering and Applied Science

(Computer Science)

Requirements for a

Software Maintenance Support Environment

Thesis submitted for the degree of

Master of Science

Amaury SIMON

4th October 1991

Thesis
1991/SIM

Abstract

This thesis surveys the field of software maintenance, and addresses the maintenance requirements of the Aerospace Industry, which is developing huge projects, running over many years, and sometimes safety critical in nature (e.g. ARIANE 5, HERMES, COLUMBUS). Some projects are collaborative between distributed European partners.

The industry will have to cope in the near and far future with the maintenance of these products and it will be essential to improve the software maintenance process and the environments for maintenance.

Cost effective software maintenance needs an efficient, high quality and homogeneous environment or Integrated Project Support Environment (IPSE). Most IPSE work has addressed software development, and has not fully considered the requirements of software maintenance.

The aim of this project is to draw up a set of priorities and requirements for a Maintenance IPSE. An IPSE, however can only support a software maintenance method. The first stage of this project is to define 'software maintenance best practice' addressing the organisational, managerial and technical aspects, along with an evaluation of software maintenance tools for Aerospace systems. From this and an evaluation of current IPSEs, the requirements for a Software Maintenance Support Environment are presented for maintenance of Aerospace software.

Acknowledgements

This thesis is dedicated to my wife and parents.

I would like to thank them for the support and encouragement that they have given me.

I am grateful to my supervisor Professor Keith BENNETT and to Jean-Paul DENIER for their encouragement and guidance throughout this study.

This work was supported by MATRA-ESPACE.

Contents

1 Introduction 1

1.1 Purpose of the research 1

1.2 Objectives of the research 2

1.3 Thesis Structure 2

2 What is Software Maintenance ? 4

2.1 Introduction 4

2.2 Software Maintenance Activities 4

2.2.1 Introduction 4

2.2.2 Software Maintenance 5

2.2.3 Corrective Maintenance 6

2.2.4 Adaptive Maintenance 7

2.2.5 Perfective Maintenance 8

2.2.6 Preventive Maintenance 9

2.2.7 Conclusion 11

2.3 Maintenance and the Software Life-cycle 12

2.3.1 Introduction 12

2.3.2 The Software Life-cycle 12

2.3.3 Criticisms of the Classical Life-cycle 13

2.3.4 Conclusion 14

2.4 Maintenance Problems 15

2.4.1 Introduction 15

2.4.2 Activity 15

2.4.3 Process 16

2.4.4 Quality 16

2.4.5 Software 19

2.4.6 Maintainability 19

2.4.7 Documentation 20

2.4.8 Users 21

2.4.9 Staff 22

2.4.10 Conclusion 22

2.5 The Economics of Software Maintenance 23

2.5.1 Introduction 23

2.5.2 Software Maintenance Costs 23

2.5.3 Software Maintenance Cost Estimation 26

2.5.4 Conclusion 30

2.6 Summary 30

3 Software Maintenance Best Practice **31**

3.1 Introduction 31

3.2 The Organisational side of Software Maintenance 33

3.2.1 Introduction 33

3.2.2 Strategy for implementing software maintenance process change 33

3.2.3 Organising the maintenance activity 35

3.2.4 Organising the maintenance of systems 38

3.2.5 Conclusion 40

3.3 The Management side of Software Maintenance 41

3.3.1 Introduction 41

3.3.2 Planning for maintenance 41

3.3.3 Monitoring and controlling maintenance 43

3.3.4 Tools for maintenance management 46

3.3.5 Organising the maintenance department 48

3.3.6 Managing the maintenance team 53

3.3.7 Conclusion 56

3.4 The Technical side of Maintenance 57

3.4.1 Introduction 57

3.4.2 Software Maintenance Models 57

3.4.3 Software Maintenance Model for Aerospace Systems 63

3.4.4 Technical Information for Maintenance Staff 71

3.4.5 Conclusion 73

3.5 Summary 74

4 Software Maintenance Tools **75**

4.1 Introduction 75

4.2 Commercially Available Tools 76

4.2.1 Tools for Program Comprehension 76

4.2.2 Tools for Reverse Engineering 82

4.2.3 Tools for Testing 85

4.2.4 Tools for Maintenance Management 89

4.3 Prototypes and Research Projects 91

4.3.1 Prototypes for Program Comprehension 91

4.3.2 Prototypes for Fault Localisation 93

4.3.3 Prototypes for Impact Analysis 95

4.3.4 Prototypes for Knowledge-based Systems in Maintenance 97

4.4 Summary 99

5 Integrated Project Support Environments **100**

5.1 Introduction 100

5.2 What is an I.P.S.E. ? 100

5.2.1 Introduction 100

5.2.2 Features of an IPSE 101

5.2.3 The Ideal IPSE 102

5.2.4 Conclusion 103

5.3 Requirements for Software in the Aerospace Industry 104

5.3.1 Introduction 104

5.3.2 Safety Critical Systems 104

5.3.3 Increasing Software Size 104

5.3.4 Increasing System Lifespan 105

5.3.5 Distributed Developments 105

5.3.6 System Perenniality 106

5.3.7 Reuse 106

5.3.8 Training and Knowledge Transfer 107

5.3.9 Conclusion 107

5.4 Criteria for Analysing IPSEs 108

5.4.1 Introduction 108

5.4.2 Flexibility 108

5.4.3 Integration 109

5.4.4 Distribution 110

5.4.5 Conclusion 110

5.5 Evaluation of IPSEs 111

5.5.1 Introduction 111

5.5.2 Evaluation of IPSEs with the criteria 112

5.5.3 Conclusion 117

5.6 IPSEs for Aerospace Systems 118

5.6.1 Introduction 118

5.6.2 HERMES/COLUMBUS 118

5.6.3 FREEDOM 119

5.6.4 Evaluation of Aerospace IPSEs 120

5.6.5 Conclusion 121

5.7 Summary 121

6 Requirements for a Software Maintenance Support Environment **122**

6.1 Introduction 122

6.2 Data Base 123

6.3 Human Computer Interaction . . . 123

6.4 Software Configuration Management . . . 124

6.4.1 Software Configuration Identification . . . 124

6.4.2 Software Version Control . . . 124

6.4.3 Software Change and Configuration Control . . . 125

6.4.4 Software Configuration Status Accounting . . . 125

6.4.5 Software Configuration Audit . . . 126

6.5 Program Comprehension . . . 126

6.5.1 Static analysis . . . 126

6.5.2 Dynamic analysis . . . 128

6.5.3 Impact analysis . . . 129

6.5.4 Traceability . . . 129

6.6 Quality Assurance . . . 130

6.7 Planning and Controlling maintenance . . . 130

6.8 Distribution . . . 130

6.9 Others . . . 131

6.9.1 Reuse . . . 131

6.9.2 Reverse Engineering . . . 131

6.9.3 Safety Critical Systems . . . 131

6.9.4 Environment simulator . . . 131

6.9.5 Documentation for Maintenance 131

6.9.6 Training 132

6.9.7 Knowledge Transfer 132

6.10 Summary 132

7 Conclusions and Further Research **133**

7.1 Conclusions 133

7.2 Further Research 135

A Software Maintenance Tools Commercially Available **137**

A.1 Tools for Program Comprehension 137

A.1.1 Code Analyser 137

A.1.2 Code Visualisation 147

A.1.3 Cross Referencer 150

A.1.4 Source Code Comparison 152

A.1.5 Execution Monitoring /Debugging 152

A.2 Tools for Reverse Engineering 155

A.2.1 Restructurer 155

A.2.2 Reformatter 155

A.2.3 Reengineering 156

A.2.4 Reverse Engineering 157

A.3 Tools for Testing 160

A.3.1 Test Coverage Monitors 160

A.3.2 Regression Testing 164

A.4 Tools for Maintenance Management 170

A.4.1 Software Configuration Management 170

A.4.2 Program Synthesis 177

A.4.3 Library Management 178

A.4.4 Change Management 178

A.4.5 Change Control 179

A.4.6 Version Control 179

A.4.7 Product Management 181

B Software Maintenance Prototypes and Research Projects **183**

B.1 Prototypes for Program Comprehension 183

B.1.1 Code Analyser 183

B.1.2 Program Understanding 185

B.1.3 Knowledge Based System and Maintenance Assistant 185

B.2 Prototypes for Fault Localisation 192

B.2.1 Fault Detection 192

B.2.2 Fault/Error Localisation 193

B.3 Impact Analysis 196

B.3.1 Dependency Analysis 196

B.3.2 Ripple Effect Analyser 196

B.4 Management Prototypes 198

B.4.1 Software Configuration Management 198

B.4.2 Inverse Software Configuration Management 205

B.4.3 Product Management 205

B.5 Environment Prototypes 206

B.5.1 Programming Environment 206

B.5.2 Software Maintenance Environment 210

Chapter 1

Introduction

1.1 Purpose of the research

The Aerospace industry is concerned with huge software projects, sometimes safety-critical in nature, containing millions of lines of code, whose development times are typically of the order of several years. These projects are collaborative between distributed European partners.

The industry will have to cope in the near and far future with the maintenance of these products and improve the software maintenance process and the environments for maintenance. There has been much research on software development environments and some of them are commercially available, but environments for maintenance have not been addressed in full.

Cost effective software maintenance needs an efficient, high quality and homogeneous environment (or IPSE). An Integrated Project Support Environment is an integrated environment that focuses on the developmental aspects of the software life-cycle. Most IPSE work has addressed software development, and has not considered the fully requirements for software maintenance.

The purpose of this research is to draw up a set of priorities and requirements for a Software Maintenance Support Environment that could be used in the Aerospace industry.

1.2 Objectives of the research

Software maintenance is usually the most expensive phase of the software life-cycle [151], and there is a lack of good maintenance practice as well as environments for maintenance in most industrial and commercial organisations.

The work in this thesis is firstly concerned with identifying the best way to cope with software maintenance by defining 'software maintenance best practice' based on current practice and analysis of the maintenance problem.

Software maintenance best practice is addressed at different levels: organisational, managerial and technical.

The organisational level is concerned with the adoption of the best strategy for this activity and for the software products. The current software maintenance process has to be analysed to reveal its weaknesses in order to define a better software maintenance process. The maintenance department has to be organised so that it can become more efficient and productive.

The management level is concerned with the best ways to manage, plan and control the software maintenance process, and to organise and manage the maintenance department.

The technical level is concerned with the different tasks in the software maintenance process and the technical information needed to perform maintenance.

From the best method to perform maintenance, a survey on software maintenance tools and an evaluation of current IPSEs, the requirements for a Software Maintenance Support Environment are presented.

1.3 Thesis Structure

The second chapter describes software maintenance in terms of its different activities. The place allocated to software maintenance within the software life-cycle is evaluated, the problems with the maintenance activity are listed and the economics of software maintenance are investigated.

The third chapter defines 'software maintenance best practice' on the basis of maintenance problems enumerated in the previous chapter and analysis of current practice. Software maintenance best practice is separated into three categories:

- organisational
- management
- technical

The fourth chapter presents a survey on software maintenance tools that can be utilised for Aerospace systems. This survey is divided into commercially available tools and prototypes and research projects.

The fifth chapter surveys and assess current IPSEs according to requirements in the aerospace industry.

The sixth chapter specifies the requirements for a Software Maintenance Support Environment based on software maintenance best practice and an evaluation of software maintenance tools and current IPSEs.

The seventh chapter contains conclusions and further research.

Chapter 2

What is Software Maintenance ?

2.1 Introduction

The objective of this first chapter is to define software maintenance and to explain the place for maintenance in the software life-cycle. We identify the major maintenance problems and the cost of software maintenance itself.

2.2 Software Maintenance Activities

2.2.1 Introduction

Software maintenance is a complex and serious problem, serious because of the costs, and complex because of the wide range of activities involved e.g. *requirement analysis, error diagnosis, program comprehension, impact analysis, solution analysis, software changes, test and simulation, repair or installation, change control, and quality assurance.*

This section outlines software maintenance in terms of its different activities.

2.2.2 Software Maintenance

There is a growing interest in software maintenance as seen in the number of articles, reports and textbooks on the subject, and this field has become established as a sub-discipline within the general field of software engineering. Software maintenance has traditionally been seen as the final phase of the software life-cycle and given low priority whereas the development phases of *requirement, design, code, testing* have been given greater prominence.

The term 'software maintenance' is now well established in the computing profession and industries, but in many ways it is an unfortunate choice of words, suggesting parallels or similarities with hardware maintenance. However, hardware maintenance is usually required because of the progressive degradation or wearing out of physical materials, while software is not subject to such factors.

Furthermore 'maintenance' carries undesirable connotations for many people, implying that some rather low-level, unintellectual activity is being undertaken. Instead of maintenance, other terms have been employed like **enhancement, support, further development, program evolution** or **logistic support**. Often, software maintenance refers only to debugging. In this report, we shall use the wider ANSI-IEEE definition [110]:

> *Software maintenance is the 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.*

It is the set of activities which result in change to the originally accepted product set. The changes arise because of modifications created by correcting, inserting, deleting, extending, and enhancing the *baseline system.*

The baseline [12] is the foundation for configuration management (see section 4.2.4- 1). It provides the official standard on which subsequent work is based and to which only authorised changes are made. After an initial baseline is established and frozen, every subsequent change is recorded as a delta until the next baseline is set.

Software maintenance activities have been divided into three categories by Swanson in 1978

[150]: corrective, adaptive and perfective. These terms have been widely adopted in the industry and form a useful distinction in classifying types of software maintenance.

Whereas *corrective maintenance* refers to a changes usually triggered by a *failure* of the software detected during operation, *adaptive* and *perfective maintenance* refers to changes due to user requests. These terms are defined in section 2.2.3-4-5 below. Some authors (Swanson, 1976; Glass and Noiseaux, 1981; Arnold and Parker; Pressman, 1987; Pfleeger, 1987; Gamalel-Din and Osterweil, 1988) [256, 90, 8, 198, 196, 85] refer to an additional form termed *preventive maintenance* which is the work that is done in order to try to anticipate and prevent malfunctions or improve quality attributes in particular *maintainability*.

Lientz and Swanson have administered a survey (1980) to determine how much time each type of maintenance activity requires [151]:

Perfective Maintenance	50%
Adaptive Maintenance	25%
Corrective Maintenance	21%
Preventive Maintenance	4%

Also, we shall refer to another type of maintenance, *Slum clearance*, which is the extreme case of software maintenance when the software can no longer be maintained, or at least the cost of imposing any change would exceed the cost of complete replacement. Slum clearance can be seen as the **termination** or **retirement** of the software.

In the Aerospace industry, the term *evolutive maintenance* is used and refers to any effort which is initiated as the result of modifications in the mission according to changing needs or requirements. This can be seen as enhancement according to different authors and perfective maintenance according to above definition.

2.2.3 Corrective Maintenance

Corrective maintenance refers to changes necessitated by actual *errors* in a system. It consists of activities normally considered to be error correction, required to keep the system operational and that must often be corrected immediately.

The terms error and fault for a specific defect within a system are usually used. Anderson [5] defined an *error* as a part of an erroneous state which constitutes a difference from a valid state, and an error in a component or the design of a system as a *fault* in the system. A component fault in a system is an error in the internal state of a component, and a design fault in a system is an error in the state of the design.

The fault is manifested in software deviating from its intended function. Examples of errors or faults include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification.

Corrective maintenance is needed as the result of three main causes [178]: design, logic and coding errors.

1. Design errors are generally the result of erroneous or incomplete design. When a user gives incorrect, incomplete, or unclear descriptions of the system being requested, or when the analyst/designer does not fully understand what the user is requesting, the resulting system will often contain design errors.

2. Logic errors are typically the result of invalid tests and conclusions, faulty logic flow and incorrect implementation of the life-cycle steps, and are usually attributable to the designer or earlier maintenance work. Often logic errors occurs when unique or unusual combinations of data, which were not tested during the development or previous maintenance phases, are encountered.

3. Coding errors are the result of either incorrect implementation of the detailed logic design, or the incorrect use of the source code. These errors are caused by the programmer; they are usually errors of negligence and are the most inexcusable.

2.2.4 Adaptive Maintenance

Adaptive maintenance involved any effort which is initiated as a result of changes in the environment in which a software system must operate. This maintenance activity is performed in order to make the software product usable in the changed environment.

For example, new versions of the operating system may be introduced, or the software may be moved to new or different hardware. These environmental changes are normally beyond the control of the software maintainer and consist primarily of change to the:

- system software, e.g., operating systems, compilers, utilities
- hardware configurations, e.g., new terminals, local printers
- data formats, file structures

Changes to operating system software (compilers, utilities, etc) can have varying effects on the existing application systems. These effects can range from requiring little or no reprogramming, to simply recompiling all of the source code, to rewriting code which contains non-supported features of a language that are no longer available under the new software.

Changes to the computer hardware (new terminals, local printers, etc) which support the application system are usually performed to take advantage of new and or improved features which will benefit the user. They are normally performed on a scheduled basis. The usual aim of this maintenance is to improve the operation and response of the application system.

Changes to data formats and file structures may require extensive maintenance on a system if it was not properly designed and implemented. If reading or writing of data is isolated in specific modules, changes may have less impact. If it is embedded throughout the code, the effort can become very lengthy and costly.

Maintenance resulting from changes in the requirements specifications by the user, however is considered to be perfective, not adaptive maintenance.

2.2.5 Perfective Maintenance

Perfective maintenance includes all changes, insertions, deletions, modifications, extensions, and enhancements which are made to the system to meet the evolving and/or expanding needs of the users.

For example, a tax program may need to be modified to reflect new tax laws or a payroll program may need to be modified to incorporate a new union settlement, but usually, modifications are much more substantial.

Perfective maintenance refers to *enhancements* made to improve software functionality. It is generally performed as a result of new or changing requirements, or in an attempt to augment the software. Optimisation of the performance of the code to make it run faster or use storage more efficiently is also included in the perfective category.

Perfective maintenance is required as a result of both the *failures* and *successes* of the original system. A *failure* is the inability of a system or system component to perform a required function within specified limits. If the system works well, the user will want additional features and capabilities. If the systems works poorly, it must be fixed. As requirement change and the user becomes more sophisticated, there will be change requested to make functions easier and/or clearer to use. Perfective maintenance is the method usually employed to keep the system up-to-date, responsive and germane to the mission of the organisation.

There is a further aspect of perfective maintenance that is having a serious economic impact. In order to maintain a competitive edge, a company must prepare new products, services, etc. Often this demand changes to the company's software and there is evidence that serious delays are occurring because the software cannot be modified easily, quickly and reliably. Delays of up to two years have been reported informally, with consequent effects upon the organisation's marketing strategy. It would seem that the backlog is not simply attributable to poor project scheduling and planning; it is rather that changing existing software is a difficult and skilled task.

2.2.6 Preventive Maintenance

Preventive maintenance includes the activities designed to make the code, design and documentation easier to understand and to work with, such as restructuring or documentation up-dates. For example a section of code that has had many alterations made to it may be completely rewritten, to improve its *maintainability*.

Typically, the need for preventive maintenance is stimulated from within the maintenance organisation, although it is recognised that such a need can be a consequence of a major change request from a user, which is infeasible to implement using the software as it is.

Fine tuning existing systems to eliminate shortcomings and inefficiencies and to optimise the process is often referred to as preventive maintenance. It can have dramatic effects on old, poorly written systems both in terms of reducing resource requirements, and in making the system more maintainable and thus, easier to change or enhance.

Preventive maintenance may also include the study and examination of the system prior to the occurrence of errors or problems. Fine tuning is an excellent vehicle for introducing the programmer to the code, while at the same time reducing the likelihood of serious errors in the future.

The extreme case of preventive maintenance can be seen as Slum Clearance and the onset of this activity may be triggered by any of several things:

- The inability to maintain the support software or hardware.
- The loss of the only person who understood an undocumented program.
- The inadvertent loss of the source code (through fire, flood or lack of effective configuration management).
- Deliberate and rational decision.

Whatever the cause, the effect is that the software can no longer be maintained, or at least the cost of imposing any change would exceed the cost of complete replacement. However the only significant problem appears to be that of deciding when the phase should start (assuming it is by decision rather than by accident).

A feature of maintenance work is the degradation of the system being maintained, usually demonstrated by an increase in the difficulty of working with the system or more faults. Eventually the system must be replaced by a new system as maintenance becomes too costly or the system becomes obsolescent.

The reasons for the degradation are many, but include the fact that a larger number of people work on the system, over a longer period of time than in any other phase of development and the time available is often much shorter.

2.2.7 Conclusion

Software Maintenance has been defined in terms of categorisation of tasks and different categories of maintenance e.g. corrective, perfective, adaptive and preventive have been explained. A clear view of the different categories will be useful for the following chapters to get a better understanding of the software maintenance process.

As explained in this section, there are no common or agreed definitions for these different activities and there is some disagreement whether the addition of new capabilities should be considered maintenance or additional development. Since it is an expansion of the existing system after it has been placed into operation, and is usually performed by the same staff responsible for other forms of maintenance, we shall classify it as maintenance to conform to the IEEE definition.

2.3 Maintenance and the Software Life-cycle

2.3.1 Introduction

In the previous section, the software maintenance activities have been defined; it is now important to show the place of software maintenance in the *software life-cycle.*

2.3.2 The Software Life-cycle

The *development life-cycle* [6] is the period of time that begins with the decision to develop a software product and ends when the product is delivered. The development cycle typically includes a *requirements phase, design phase, implementation phase, test phase*, and sometimes, *installation and check out phase.*

The *software life-cycle* [6] is the period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life-cycle typically includes the development life-cycle and the operation and maintenance phase.

1. The **requirement phase** [6] is the period of time during which the requirement for a software product, such as the functional and performance capabilities are defined and documented.
2. The **design phase** [6] is the period of time during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements.
3. The **implementation phase** [6] is the period of time during which a software product is created from design documentation and debugged. Design must be translated into a machine executable form. The coding step accomplishes this translation through the use of conventional programming languages (e.g., Fortran, Cobol, Pl/1, Ada, C, Pascal) or so-called Fourth Generation Languages.
4. The **test phase** [6] is the period of time during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied.

Testing is multi-step activity that serves to verify that each software component properly performs its required function with respect to the specifications and validates that the system as a whole meets overall customer requirements. In any case, testing by means of program execution is generally achieved bottom up, first at the unit (module or procedural) level, then functionally, component by component. As tested components becomes available they are then assembled into a system in an integration process and system test is initiated.

5. The **installation and check-out phase** [6] is the period of time during which the software product is integrated into its operational environment and tested in this environment to ensure that it performs as required.

6. The **operation and maintenance phase** [6] is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changed requirements.

Once the system has been released the maintenance process begins. Maintenance is actually the re-application of each of the preceding activities for existing software. The re-application may be required to correct an error in the original software, to adapt the software to changes in its external environment, or to provide enhancement to function or performance requested by the customer.

2.3.3 Criticisms of the Classical Life-cycle

Traditionally, the maintenance phase has been regarded as not belonging to the software development life-cycle, but rather as occupying a detached position. However, it is inappropriate to regard it as a stage that is independent of the other stages of the life-cycle. Sommerville [249] states:

The maintenance activity may involve changes in requirements, design and implementation or it may highlight the need for further system testing.

So the maintenance programmer may have to perform many of the activities that have been performed during the development phases.

Lehman[141, 142] suggests that large software systems are never completed and that such systems are always being maintained. He suggests that the term 'maintenance' should be avoided and that 'program evolution' be used. Such an approach recognises that when a product is delivered to the customer it is just the first of a number of releases in the evolution of the product.

McKee [168] advocates that maintenance would be more accurately portrayed as 2nd, 3rd, ..., nth round development.

There have been many criticisms of the classical software life-cycle model [167, 89]. In particular, it has been argued that the model stresses the importance of the development stages and yet maintenance is the main software engineering activity that takes place in the lifetime of a well-used large software system.

2.3.4 Conclusion

At the time, the traditional software life-cycle model was established, software maintenance had not assumed the great importance it has today, and so the model was oriented almost exclusively to the *development* of software. Consequently software maintenance has found its niche within the model by default. The software life-cycle is product-based and the process that has created the product is not mentioned with all management activities. Therefore, there is an important area of research on the modelisation of all activities involved in the software development and maintenance process. This section has revealed the shortcomings of the traditional software life-cycle model with respect to the maintenance of software.

2.4 Maintenance Problems

2.4.1 Introduction

To attack intelligently software maintenance problems, we must know what they are in order to define the software maintenance best practice. Generally, the software maintenance problems can be categorised as organisational, managerial and technical. Most of these problems, however, can be traced to inadequate management control of the software maintenance process.

A study of the reasons of the high cost of software in 1976 reported on 24 problems areas of software maintenance; demand for enhancement and poor documentation lead the list [149]. Some of the maintenance problems are cited from a survey [178] of selected Federal and private sector ADP organisations conducted by the Institute for Computer Sciences and Technology (ICST).

This section presents different problems in software maintenance that can be seen in current practice: the activity, the process, the software, the quality, the users, the documentation, the staff and the maintainability.

2.4.2 Activity

Traditionally, software maintenance has been regarded as not belonging to the software life-cycle in the same sense as the earlier stages, but rather occupying a detached position and considered as a post-delivery activity.

The word 'maintenance' carries connotations of a less intellectual activity than 'design' because:

- Maintenance is perceived as having a low profile and is a labour-intensive activity.
- Maintenance is extremely important but a highly neglected activity.
- Many people think that Software Maintenance is just the correction of errors resident in a program when it is released.
- Many people think money spent in software maintenance is wasted.

- Maintenance is always under budgetary pressures.

Many people have a wrong idea on the software maintenance activity; it is important to clear this view.

2.4.3 Process

There is no satisfactory general method for the *software maintenance process* which implies:

- Lack of management of software maintenance.
- Lack of management visibility of software maintenance.
- Lack of understanding of how to maintain software.
- Lack of metrics
- Lack of historical data on maintenance and error histories.
- Difficulty in tracing the product or the process that created the product.
- Difficulty in estimating the cost of modifications.

The reasons suggested for this [136] was that maintenance is too domain and system specific for such a method to ever be developed. A software maintenance process should be developed and tailor to organisation and products.

2.4.4 Quality

Software quality is [6]:

1. The totality of features and characteristics of a software product that bear its ability to satisfy given user needs; for example conform to specifications.
2. The degree to which software possesses a desired combination of attributes.

3. The degree to which a customer or user perceives that software meets his or her composite expectations.
4. The composite characteristics of software that determines the degree to which the software in use will meet the expectations of the customer.

There are many definitions of software quality and none of them is perfect because it seems difficult to measure this criteria. There are maintenance problems arising from these definitions:

- During maintenance, the specification or design are rarely complete, precise or verifiable according to the software product.
- What are the desired combination of attributes ?
- The user's needs are evolving (see section 2.4.8).
- How do we measure software quality ?
- During development and maintenance there is a lack of quality management.

Software quality attributes often deteriorates with time since older systems tend to grow with age, to become less organised with change and become less understandable with staff turnover. A lack of attention to software quality during the design and development phases generally leads to excessive software maintenance cost.

It is difficult to have good quality software [178] with different software languages used, poor quality design and code and a lack of common data definitions.

- **Software languages:**

  The use of more than one programming language in an application system is often the cause of many software maintenance problems. If the maintainer is not proficient in the use of each of the specific languages, the quality and consistency of the system can be affected and interfacing them can be very tricky because often there are ad-hoc.

- **Poor software design:**

  The design specifications of a software system are vital to its correct development and implementation.

Poor software design can be attributed to [178]:

- a lack of understanding by the designer of what the user requested.
- poor interpretation of the design specifications by the developers.
- a lack of discipline in the design which results in inconsistent logic.
- no design history.
- no standards.
- no methods.

- **Poorly coded software:**

  As computer programming evolved, much of the code development was performed in an undisciplined, unstructured manner. The result of poor programming practices is:

  - few or no comments.
  - poorly structured programs.
  - use of non-standard language features of the compiler.
  - lack of Quality Assurance and metrics.

  It is even more difficult to understand poorly written code if the program has been modified by different individuals with a multiplicity of programming styles.

- **Lack of common data definitions:**

  An application system should have common data definitions (variable names, data types, data structures, etc) for all segments of the system. These common definitions entail the establishment of global variable names which are used to refer to the same data array or record should be defined and used for all programs in the system.

2.4.5 Software

Corrective, adaptive and perfective maintenance activities typically result in growth and degradation of the software's structure (this is known as program entropy). Lehman [142, 143] stated in his second law of program evolution:

As an evolving program is continually changed its complexity, reflecting deteriorating structure, increases unless work is done to maintain and reduce it.

The net result of continual modification is that software tends to increase in size and its structure tends to degrade with time. For example [99], the average program grows by 10% every year, resulting in its doubling in size every seven years. It is important to increase the maintainability of software to cope with its increasing lifespan (see section 5.3.4) in order easily to maintain it.

It is also difficult adequately to maintain software because:

- Software were created without maintenance in mind.
- A change in one component often affects another one by introducing unforeseen *side effects.*
- The average life expectancy of a software has increased from about three years, two decades ago, to seven to ten, and even more for example in the Aerospace industry (see section 5.3.4). Therefore, with the degradation of the software structure and the increasing size of the software, it will be more difficult to maintain it.

2.4.6 Maintainability

Maintainability is defined by Martin and McClure [163] as:

> The ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements.

No quantitative or contractual definition of maintainability are available. There is a need to think about the maintainability of the software at the beginning of the life-cycle, which will require extra cost that cannot be easily justified. The maintainability of software deteriorates during the software life-cycle because:

- The need to change the software have been underestimated in the development and the modifications affect the maintainability of the software.

- Maintenance changes tend to degrade the structure and quality of programs by increasing their complexity and making them more complex and more difficult to maintain next time.
- Maintainability requirements are omitted in system requirement specification [95].
- Many systems have been designed and developed without any serious consideration being given to their long term maintainability [128].

An alternative approach is to assess maintainability indirectly in terms of the software itself e.g. modularity, cohesion and coupling.

2.4.7 Documentation

Software documentation is the technical data or information, including computer listings and print-out, in human readable form, that describe or specify the design or details, explain the capabilities, or provides operating instructions for using the software to obtain the desired results from a software system.

The software documentation is one of the major problem of software maintenance because:

- Software documentation arouses very little interest amongst programmers. It is often considered to be one of those jobs that should be put off until the last opportunity and in many cases it is put off indefinitely. Perhaps this low view of documentation is perpetuated by its consistent poor quality.
- Document structure does not provide enough visibility for maintenance concerns [95]. When a maintainer has to perform his job, the only source of information available may be the documentation and the code.
- Software documentation is often inadequate, incomplete, non-existent, or out-of-date. Where it does exists, it usually consists of an unmanageable set of unstructured papers that are difficult to access and impossible to maintain. The development team has some difficulties to understand needs of maintainers and prefers to answer their own needs.
- Software documentation is written by developers who don't understand maintenance.

- Schneidewind [227] reports that it is very difficult to maintain software which was not adequately documented.
- A survey from Chapin [43] showed that poor documentation is the biggest problem in the software maintenance work.

2.4.8 Users

A critical part of effective maintenance is communications, both among development personnel and between the development and user organisations. Users have a lack of understanding on systems and are not adequately trained. User demand for enhancement and extension of their systems is a major problem for the maintenance team because:

- Users are often unable concisely to specify what they want from an application system and express requirements in a way totally different to system structure or behaviour.
- If a system does what the user needs, the user will often think of things to add. The more successful a system is [178], the more additional features the user will think of and sometimes user expectations are unrealistic.
- If a system does not work well [178], there will be a constant demand for remedial action to make it function properly.
- The user is often unaware of the impact that one change can have on both the system and the maintenance workload, and expects a rapid response to his or her needs.

2.4.9 Staff

The Lientz and Swanson studies [149] indicates low morale and productivity as a major problem during maintenance. It is difficult to motivate maintenance staff and to have a maintenance team working properly because:

- Development is seen in a company when it is complete as a successful achievement whereas maintenance will not receive any such consideration; maintenance is an on-going activity.

- Software maintenance is considered as unimportant, unchallenging, uncreative work.
- Staff have competing demands on their personnel time (poor availability of maintenance staff).
- Maintenance staff usually have not been involved in the development of the product so they have no knowledge of the process that made the product.
- Maintenance staff turnover is very high. Experienced personnel are replaced with new personnel who are unfamiliar with the applications software, and may be unfamiliar with the programming environment (tools, methods, software maintenance process ...) as well. The turnover rate is so high that there is little time allocated to update the documentation adequately.
- Maintenance is often used in a company as training for new programmers who have little software engineering or application domain knowledge.
- No time is spent for adequate training and there is the remaining problem of how to train the maintenance staff.

2.4.10 Conclusion

This section described the various maintenance problems in order to better understand why maintenance costs huge amount of money and often seen as a bottleneck. These problems are perceived as being organisational, managerial and technical. We need to solve these problems by defining a good method to perform the software maintenance process.

2.5 The Economics of Software Maintenance

2.5.1 Introduction

The objective of this section is to explain the economics of software maintenance with different citations and studies from the literature, to define the models for estimating the cost of maintenance and to make criticism on them.

2.5.2 Software Maintenance Costs

Since estimating software maintenance costs is very difficult, we shall provide some citations to better understand its propensity and magnitude, and to recognise that large amount of money are involved:

- **Maintenance cost in the US federal government:**

  It owns about 25 billion lines of code and is spending $3.75 billion on its Information Technology budget on maintenance (Lientz and Swanson, 1980) [151].

- **Maintenance cost in the US DoD:**

  It was estimated in 1985 that the DoD spends about three-four percent of its budget, or approximately 10 billion dollars per year on software and this number is expected to increase rapidly in the next few years [82].

- **Maintenance cost in the US:**

  It was estimated that $30 billion are been spent on maintenance in the USA annually [151] and it has been reported that the United States spends 2% of its GNP on software maintenance.

- **Maintenance cost Worldwide:**

  Martin and McClure say that in 1983 more than $ 30 billion per year are being spent on maintenance worldwide [163].

- **Hardware/Software cost trend:**

  Boehm [29], first presented the Hardware/ Software cost trends diagram that predicted a dramatic rise in software cost relative to Hardware cost at a time when spending more on software than Hardware was difficult for many to perceive.

- **DoD software market share:**

  The total software budget of the DoD was estimated to be 5 % of the market, and to include $ 10 billion spent on embedded systems [255].

- **Maintenance programmers/programmers time:**

  There are an estimated one million programmers in the US alone and more than half the programmer's time is devoted to maintenance [187].

- **Maintenance phase/Life-cycle cost:**

  A survey by Lientz and Swanson reported that more than 50 % of the budget is spent on the maintenance phase in (1983) and Hager says that 60 % of the software costs associated with the design, development and implementation of computer systems occurs in the maintenance phase [96].

- **Development/Maintenance cost:**

  On a relative comparison between development and maintenance cost, it was estimated that one US Air Force System cost $30 per instruction to develop and $4000 per instruction to maintain over its life-time (Boehm, 1975) [30].

- **Real-time/Business applications costs:**

  Comparing the relative costs of a source instruction in the development and maintenance phase, it appears [144] that maintenance of real-time applications is proportionally much higher that the maintenance of business applications (see table No 2.1).

Application	Real-time S/W	Business S/W
Development	n	n
Corrective Maintenance	11.75*n	5*n
Adaptive Maintenance	5*n	3.5*n
Perfective Maintenance	5*n	2.5*n
Evolutive Maintenance	5*n	2*n
All mixed Maintenance	6.75*n	3.25*n

Table No 2.1: Relative cost of a source instruction

- **Projection of the cost in the 90's:**

  Pfleeger [196] said in in 1987 that the trend towards higher maintenance costs is expected to continue, and three-quarters of the system's cost is likely to be devoted to maintenance by the 1990's (see table No 2.2).

% of the software budget	1970s	1980s	1990s
Development	60-65%	60-40%	20-30%
Maintenance	35-40%	40-60%	79-80%

Table No 2.2: Escalating maintenance costs (Pfleeger 87; Pressman 87)

Although some of these figures must be treated with great scepticism, there is a wide agreement that software maintenance is a huge consumer of resources. Although software maintenance expenditure is hard to quantify, most companies would consider that they spend far too much effort on software maintenance.

With such a large proportion of the total software expenditure being spent on software maintenance, this area has the greatest potential of any in the life-cycle for reducing the overall system costs. The direction of money into the maintenance of existing systems has caused new developments to be postponed due to lack of financial and personnel resources. Any freeing of money from software maintenance, by increasing maintenance productivity, would help reduce the development backlog created by these resources shortages.

Although new development have traditionally been the focus of attention for the software community, these economics realities are drawing increasing attention to software support activities. The situation is especially acute in the DoD, where some agencies are predicting that maintenance requirements will more than double over the next five years. This problem results from the expected inflow of massive new systems, and the slow retirement of old systems. 'Getting the requirements right' is only transitory; they can only be 'right' at a given point in time.

2.5.3 Software Maintenance Cost Estimation

An estimation model for computer software uses empirically derived formulae to predict data that are required. The empirical data that support most models are derived from a limited sample of projects. For this reason no estimation model is appropriate for all class of software and no model can fully reflect the product's characteristics, the development environment and the many relevant personnel considerations. Software cost models should be used with care. If they are not calibrated to the specific organisation's experience. Models should thus be used to augment the estimation process and not replace it.

Software maintenance cost estimation models have largely been based on simple linear ratios depending on the size of the product, its development cost and the number of instruction changed. This section will define the maintenance/development ratio, Boehm's and Belady's model and the multiplicity of factors dependent on the software maintenance cost. In order to understand these models and ratios, we need to explain terms used:

Terms	Explanation
MM	man-months
(MM)mo	overall maintenance cost in man-month
(MM)m.year	maintenance cost in man-month per year
KDSI	thousands of delivered source instruction

1. **The Maintenance/Development cost ratio:**

   The Maintenance/Development cost ratio (M/D) is used to estimate the overall life-cycle maintenance cost (MM)m dependent of the development cost (MM)d:

$$(MM)m = (M/D)(MM)d$$

For example, a program has 32000 delivered source instructions and this product required 200 man-months to develop.

If we consider a (M/D) ratio of 1.5 (i.e. 60 % time of its software life-cycle is spent for maintenance), it would require and estimated of 300 man-month for maintenance:

$$(MM)m = (1.5)(200) = 300 \text{ MM}$$

If we consider a (M/D) ratio of 2.0 (i.e. 66 % time of its software life-cycle is spent for maintenance), it would require and estimated of 400 man-month for maintenance:

$$(MM)m = (2.0)(200) = 400 \text{ MM}$$

2. **The annual maintenance effort:**

Boehm [33] developed a model COCOMO (COnstructive COst MOdel) to estimate annual software maintenance in terms of a quantity called Annual Change Traffic (ACT) It is the fraction of the software product's source instructions which undergo change during a (typical) year, either through addition or modification.

For example suppose that the 32KDSI product had 4000 DSI added and 2400 DSI modified during its first year of maintenance. Then

$$ACT = (4000 + 2400)/32000 = 0.20$$

The COCOMO equation for estimating basic annual maintenance effort (MM)am, given the estimated development effort (MM)d, is:

$$(MM)m.year = (ACT)(MM)d = (0.2)(200) = 40 \text{ MM}$$

This model gives no more than a rough approximation to maintenance costs. However, it serves as a basis for computing a more accurate figure.

Belady [18] developed a formula from studies that reflected the factors involved in the maintenance cost:

$$M = p + K^{*}\exp(c - d)$$

where:

M is the total maintenance effort,

p represents productive effort,

c is a measure of the complexity caused by the lack of structured design and documentation,

d is the degree of familiarity from the maintenance team with the software,

K is an empirical constant depending on the application.

This model indicates that the maintenance effort can increase exponentially if a poor software development approach was used, and the person that used the approach is not available for the maintenance team.

These two models do not reveal the true maintenance costs because they are only dependent on few parameters (see next section). Furthermore, the estimation of the maintenance cost is problematic for many reasons:

- Maintenance costs can fluctuate considerably from project to project and for a given project from year to year (user requests, change in environment are not easily predictable).
- Degradation of the software quality during the software life-cycle will have an adverse effect on maintenance effort

3. **The software maintenance cost:**

The software maintenance cost is not only dependent on the size of the product, its development cost and the number of instruction changes, it depends on a multiplicity of factors and some of them are not easily quantifiable:

(a) Factors dependent on the software:

- size of the system
- number of delivered source instruction
- programming language
- lifetime of the system
- age of the system
- number of release
- system stability
- system reliability

- software complexity
- software modularity
- performance constraints
- history of the software
- quality and quantity of documentation
- quality of tests

(b) Factors dependent on the staff:

- availability of staff
- experience of the staff in the system
- senior/junior staff
- participation of the maintenance team in the system development process
- staff turnover
- organisation of the maintenance department
- staff under pressure because of user's requests
- user's knowledge of the system
- user's request quality

(c) Factors dependent on the application type:

- application type (Business/Scientific/Real-time)
- number of sites
- development cost
- tools, methods and environments used

Models for cost estimation provide a means of maintenance cost prediction (which is better than nothing), but they need to be tailored to the application domain and specific project.

2.5.4 Conclusion

The software maintenance cost is probably around 65% and 85% of all software cost. Because of the multiplicity of factors depending on the software maintenance cost, a model to predict it is very difficult to elaborate as there is no underlying theory to be based on. One of the best way to evaluate the overall maintenance cost or a change request is to tune the cost model used to the organisation and to provide an automated calibration based on actual product and change request histories.

2.6 Summary

The main points in this chapter have stressed the need to understand the nature, the cost and the problems of software maintenance. As explained in this chapter, maintenance is the most expensive phase of the software life-cycle, difficult to cost and carries many unsolved problems for companies. The escalating maintenance costs need to be stopped and decreased by defining a a good practice for software maintenance along with adequate maintenance tools. To make this activity more productive, a 'software maintenance best practice' report is presented in the next chapter.

Chapter 3

Software Maintenance Best Practice

3.1 Introduction

The issue that is addressed in this chapter is to reduce the software maintenance cost in most organisations by improving the software maintenance process and providing a good method for maintenance. This leads to better productivity, lower application backlog and clearer management visibility. The method advocated in this thesis is based on an analytical approach of the software maintenance process instead of experiments. Software maintenance best practice is defined from current practice and maintenance problems, and is addressed at different levels: organisational, managerial and technical, in order to solve these problems within a company. Because the software maintenance process will be improved according to criteria defined below and established by the senior management, this method can be stated as best practice.

This chapter proposes a model within which the managerial and technical issues can be presented and discussed:

The organisational level is concerned with the finances of the maintenance activity, communication with upper management and the adoption of the best strategy for this activity and for the software products to be maintained. The current software maintenance process has to be analysed to reveal its weaknesses in order to correct them and define a better software maintenance process.

The management level is concerned with the best ways to manage, plan and control the software maintenance process, and the organisation and management of the maintenance department in a more efficient and productive manner.

The technical level is concerned with the different tasks in the software maintenance process and the technical information needed to perform maintenance.

3.2 The Organisational side of Software Maintenance

3.2.1 Introduction

The organisational level is concerned with the adoption of the best strategy for the maintenance activity and for the software products to be maintained. The current software maintenance process has to be analysed by the senior management to reveal its weaknesses in order to define a better one and avoiding some maintenance problems within a company. Special attention needs to be given to the organisational role of the maintenance organisation, particularly with regard to its financial aspect. As stated in section 2.4.2, 'the maintenance activity is perceived as having a low profile, being a labour intensive activity, extremely important but highly neglected ...' thus, there is a need to change this statement by providing suitable advice for the maintenance organisation.

The organisational view of software maintenance is generally of critical importance to the success of this activity, and yet there is a lack of a general and agreed model for describing the software maintenance process in such a way that different teams can be compared under a standard paradigm.

Pressman [198] pointed out that 'there are almost as many organisational structures for software maintenance as there are for software development'. In the case of software maintenance, however, formal organisation rarely exist, but it seems valuable to give information on adopting a good strategy to obtain an appropriate maintenance organisation.

3.2.2 Strategy for implementing software maintenance process change

As explained by Humphrey in his book 'Managing the Software Process' [109] most organisations can improve their software maintenance process and need to change their organisation structure, their management system, their procedures, their tools and methods, but the implementation of the necessary changes must be handled with care (if it is too didactic, they will face a general resistance from employees).

The major changes to the software maintenance process must start at the management level in the company and everyone must be involved. Effective changes by the senior management requires knowledge of the current software maintenance process.

A descriptive approach of the current software maintenance process must be observed and assessed in real life in order to learn how the software organisation actually works. A list of major problems within the organisation must be edited and it is important to collect information on the current software maintenance process in terms of statistics.

Some indicators of less than adequate good practice could include:

- high staff turnover
- lack of metrics, statistics and visibility on software maintenance
- lack of historical data and error history on maintenance
- absence of quality plan, maintenance plan
- no use of configuration management and versioning tools
- inadequate or unused documentation

It seems evident that in many organisations, current practice differs substantially from potential best practice. Colter [56] has argued that best practice involve software maintenance as *product support*. Corrective, adaptive and perfective maintenance require a product support organisation to receive problem report, distribute upgraded versions of the software and keep customers informed of problems, solutions and new versions. This concept introduces a number of notions including the strong relationship of software maintenance to the need of the business, but also to gather round the software product a group of people committed over a number of years to its successful evolution.

The product is expected to undergo a series of evolutionary steps over its lifetime, and this is planned from the outset, including the recognition of the need for business justification for doing so.

3.2.3 Organising the maintenance activity

The organisation of the maintenance activity by senior management starts by developing a strategy and identifying this activity. The organisational role of the maintenance activity [22] need to be given particularly in financial terms. The approach advocated is to justify the added value provided by software maintenance, and thus the financial benefits to be gained by investment in the maintenance process (for example, by buying better technology). This may require a concurrent change from maintenance as merely a survival activity, in which employees are expected to meet the on-going change load without frequent serious problems, to maintenance as a product management activity [56, 57].

1. **Develop a maintenance strategy**

   Strategy is an over-used word and we need to define what it implies. The strategic analysis is an understanding of the strategic position of the organisation, e.g. the environment, value expectations, objectives and resources. The strategic choice is the formulation of possible courses of action, their evaluation and the choice between them, e.g. generation of options, evaluation of options and selection of strategy. The strategic implementation is planning how the strategy can be put into effect, e.g. resource planning, organisation structure, people and system. The strategy for maintenance should be linked to the overall strategy of the company (for example providing software of good quality to customers). To develop a good strategy for this activity, it is necessary to save the 'savoir-faire' and to understand the importance of the software to be maintained.

   - **Who will perform maintenance ?**

     At the stage of how to organise maintenance, a decision should be made as whether maintenance will be performed in-house, by the customer, or by a third party.

   - **Preserving the know-how**

     A company with a long established software development culture often find it difficult to adopt new methods, tools and procedures. Yet, management at these companies recognises that the old culture is ineffective and in some cases detrimental to the overall strategic objectives.

     The software maintenance strategy need to protect the extremely valuable corporate 'know-how' that is tied up in existing software.

- **Importance of the software for the organisation:**

  The strategy for the maintenance activity needs to consider the quality of the software that are maintained within the company. If a software is very important both at present and in the future to the organisation, and is of high quality, this would suggest continued enhancement of the existing system. A similarly important package of very low quality may merit rewriting or reengineering. Even very coarsely graduated metrics can be of considerable help to the decision process, especially if there is strong clustering.

2. **Identify the market trend**

   In order to determine the market trend for maintenance, the organisation should define its objectives according to the maintenance activity, understand the differences with the development activity, prove that with a good strategy for maintenance advantageous return on investments can be obtained and reduce maintenance costs.

   - **Objectives of the maintenance activity**

     The aim of software maintenance is to extend the life of the product as much as possible, whereas initial development requires the project to be completed within budget and on time. If the overall strategy of the company is to provide software that are highly maintainable and inspire customer confidence and satisfaction, the maintenance strategy can be to keep this level of maintainability and to continualy satisfy the customer with the software.

     The objectives of the maintenance activity have to be defined within the company in relations with other activities. It is essential that a bridge is established between the corporate strategy and the maintenance strategy.

   - **Compare Development and Maintenance**

     Because more programs are developed and place into the operation phase, a direct effect is the growing needs for maintenance and an increasing importance of the maintenance activity within the data processing organisations.

     - **Development:**

       The initial development of software is usually project based; it is undertaken to a budget and timescale; there is (hopefully) a clear product defined through requirements analysis; the project exists because of an identified market (or other) need; and the organisation may have submitted a competitive tender to win the work.

Prime objectives are expressed mainly in terms of functional and performance attributes of the software.

- **Maintenance:**

  In contrast, software maintenance is usually revenue based; in financial terms it is seen as a continuing consumer of resource with a nebulous and unquantified benefit to the organisation.

  The development of a system requires to be considered as product based in order to provide a better credibility with customers. Thus, the company develop a system, is able to maintain it and to provide support throughout its lifespan with a comprehensive plan.

- **Development and Maintenance costs:**

  Development and maintenance are generally separated budgets managed by different groups. It is important to know what are the current annual development costs and what are the current annual maintenance costs in order to compare them.

o **Prove that maintenance can give a good return on investment**

Maintenance is the biggest business in the software industry and it is totally out of control (COLTER 1988) [57] we need to prove that maintenance can give a good return on investment.

A major problem faced by the maintenance community is the lack of recognition by senior management of the problems of maintenance (see section 2.4.2) and the benefits and returns on investment that maintenance work provides. The reason behind this is the lack of effective communication between managers of the maintenance team and the senior management responsible for running the business.

It is necessary for the maintenance manager is to talk the 'language of business' in order to present their case effectively [57]. This, by necessity, required the measurement of the processes involved and the provision of statistics for consumption by managers with a justification of the added value provided by an optimised software maintenance process. The key parameter for the organisation's board of directors is money and profit whereas for the maintenance manager it may be quality or time for completing a task.

Example: If we consider economics, it will wise to pay, say 10 percent more for the development of reliable software with proper documentation. This will more than pay for itself in the long run through reduced maintenance and ease for redesign.

The problem is that development and maintenance are separated budgets managed by different groups. Thus it is difficult for the development group to negotiate a 10 percent price increase even if it will represent a 20 percent price decrease later on in the maintenance budget. It is important in the initial planning to deal with total life-cycle cost and to provide a maintenance plan; this can be addressed by senior management.

- **Reduce maintenance cost.** In reference to section 2.5.2, software maintenance is a huge consumer of resources, therefore, it is important to find methods of reducing the cost of maintenance, perhaps more important than finding new methods of developing software as existing software is going to be with us for the foreseeable future.

  A significant reduction in the maintenance costs can be realised with a design for change philosophy integrated into the engineering life cycle. By carefully identifying the expected changes to a system and rigorously applying the concepts of *information hiding* and *abstraction of interface*, the changeable aspects of a system can be isolated [96].

3.2.4 Organising the maintenance of systems

1. **Maintenance of existing systems**

   A maintenance plan is essential for each product with replacement, retirement and new release taking into account the quality and the importance of the software.

   During the maintenance process faults will be observed, reported and corrected and where appropriate, repairs to the code, the specification, the design and to the documentation will be authorised.

   Pfleeger [196] notes that the maintenance team is always involved in balancing one set of goals with another. Conflict arises between system availability for users and implementation of modifications, corrections and enhancements. Another conflict arises whenever a change is necessary. Often, a problem may be fixed in one of two ways: a quick but inelegant way that works but do not fit in with the design or coding strategy of the system, or a more involved but elegant way that is consistent with the guiding principles used to generate the role of the system. Maintainers may be forced to compromise elegance and design principles because a change is needed immediately. When, such compromise is made, several events are likely to make system maintenance more difficult in the future.

   The particular strategy adopted in any instance will depend on:

- the nature, criticality and the severity of the fault
- the size and difficulty of the change required
- the age of the software
- the structural complexity
- the intermodule coupling
- the historical rate of modifications
- the number and the nature of the program installations
- user organisations.

And this strategy will have a profound impact on:

- the rate of system complexity growth
- the life cycle cost of the system
- the life expectancy of the system

Harrison [100] has developed a model of software maintenance to determine whether a given module can be effectively modified or if it should instead be rewritten. This model suggest early identification of change prone modules through the use of change measures across release cycles.

2. **Maintenance of future systems**

   With the traditional software life-cycle, we must give the priority to the maintenance phase and establishing development methods to support maintenance.

   It seems [162] that the concepts of information hiding, modularisation and abstraction of interfaces are more maintainable than classical programming languages.

3. **Links between existing and future systems**

   As Parikh [187] pointed out, there is a gap between modern technologies used for developing systems and maintenance of old, unstructured software systems.

   A company needs to use bridge technologies [1] to help transition from maintenance using obsolescent methods and tools to maintenance using modern practices. Without technologies like Inverse Software Configuration Management [129], Reverse Engineering, the software community will be destined to manage to diverging set of tasks:

- Maintaining old code with archaic methods.
- Creating new products in advanced environments.

3.2.5 Conclusion

In this section a strategy for maintenance has been defined, the ways to identify the market trend for the maintenance activity and a strategy for implementing a change in the software maintenance process. The main problems for the organisational side of the maintenance activity are that senior management should be involved to define the priorities and objectives of this activity in accordance with the overall strategy of the company. The need for effective communication has been outlined. Therefore, the management side of the maintenance process has to be defined in accordance to the maintenance organisation.

3.3 The Management side of Software Maintenance

3.3.1 Introduction

The general principles of management are well defined and understood, allowing projects to be completed on time and within budget, but there seems to be resistance [274] to applying these in the maintenance field because maintenance is revenue-based and also poses special problems to a manager [123].

A more diverse group over a longer period of time, work on the software, with fewer defined work standards or methods, than in any of the other phase in the software life-cycle. A large proportion of time consists of trying to respond rapidly to change request due to the direct impact on the customer, so the maintenance activity takes on a fire fighting role. This role causes the backlog of less urgent requests to increase, and rules out any more controlled preventive maintenance work with a view towards reducing problem areas.

Compared to maintenance organisation defined in the previous section, the objectives of this section are concerned with adequately managing the software maintenance activity: the software maintenance process must be planned, monitored and controlled, and the maintenance department organised and managed in an efficient way.

3.3.2 Planning for maintenance

Planning for maintenance has the particular problem that it is very hard to estimate the likely demand for work, since one needs to know how many change requests the users will make.

To overcome this task, it is important carefully to define a maintenance plan [164] and to use an appropriate tool.

Perry [193] has defined a plan of action for software maintenance with the following attributes:

- It is obtainable with existing resources.

- It will improve the productivity and quality of software maintenance.
- It will put the software maintenance group in charge of software maintenance.

Five objectives for a software maintenance plan of action are required:

1. Appoint someone responsible for the software maintenance process (see section 'organising the maintenance team').
2. Set software maintenance objectives (see section 'objective of the maintenance activity').
3. Move to the maintenance release mode method of installing a group of changes on a quarterly or semiannual basis (see section 'prove that maintenance can give a good return on investment').
4. Calculate the value added by maintenance. If the value of change cannot be quantified, then management should question the purpose of it.
5. Subject maintenance to quality control and quality assurance procedures.

These objectives depend on the importance of software for the company. A schedule [135] is a plan for when task should start and when they should finish. Each maintenance task has several properties:

- The duration is the length of time it takes for each maintenance task and for the overall software maintenance process. During planning, durations are always estimated.
- The resources needed to complete the maintenance tasks include personnel, machine time, supplies ...
- The dependencies of the task are other maintenance tasks that must finish before it starts.
- The planned completion date is the date when the task is expected to be finished.
- The promise date is the last date, the planned completion has been changed.

3.3.3 Monitoring and controlling maintenance

The high cost of maintenance can, in part, be attributed to the greater difficulty in controlling the maintenance process than processes in other phases of the life cycle.

Without proper tools, some questions remain difficult to answer:

- How long are defect fixes tacking ?
- Which programs have most defects ?
- Has this particular problem occurred before ?

These tools must:

- Store details of all problems and their solutions.
- Reports: list problems by customers, by modules, by products.

Key process parameters include:

- Response time to undertake changes
- Resource required to make a change
- The distribution of this resource across the several maintenance and Quality Assurance activities.

1. **Controlling the software maintenance process**

   The control of software maintenance can in part be realised through the use of three easily understood and implemented methods and techniques: configuration management, change control and communication [25].

   Configuration management is explained and example of tools are given in section 4.2.4.-1

Without good **communication** (see section 3.3.5.-4), control is lost, the effectiveness of maintenance procedures cannot be determined, and resource management becomes impossible. Without monitoring software quality, it will quickly degrade. It is not simply the code that requires monitoring, but documentation, design information etc.

The main ingredient of a good **change control** facility are [250]:

- a formal mechanism for identifying and communicating problems and change requirements
- reporting procedures that succinctly identify the nature, cost and timescale of the proposed change Software Problem Report, Software Change Report)
- a visible organisational structure for the approval of changes (Configuration Control Board)
- a powerful software configuration management capability to rework and control the change
- a capability to analyse change status/trends.

2. **Managing the Change**

Successful management of change is crucial for a good release of the product without uncontrolled side effects.

The steps involved [250] are:

- to evaluate the impact of the change; the affected elements need to be determined, any related element tracked and the impact established by appropriate methods
- to review the impact of the change on the project costs and timescales and decide on the strategy for implementation if accepted.
- to control the elements under change and ensure that the later versions are released in a compatible way

Change impact determination:

A configuration management system should allow enquiries to be made of its database so that all elements related to those for which the change is proposed can be established. These may be hardware, software and documentation elements. During this process, an indication should also be obtained from a good configuration management tool (see section 4.2.4.-1) to any other changes pending on any of these elements.

3. **Collecting data**

A key indicator of software quality is the defect or error rate. Poor maintenance may cause this rate to rise, due to the ripple effects of a modification. Swanson has provided a more detailed list of data that can usefully be recorded [256, 198] for each maintenance subproject:

- program identification
- number of source statements
- number of machine code instructions
- programming language used
- program installation date
- number of program runs since installation
- number of processing failures
- program change level and identification
- number of source statements added by program change
- number of source statements deleted by program change
- number of person hours spent per change
- program change date
- identification of software engineer
- change request identification
- maintenance type
- maintenance start and close dates
- cumulative number of person-hours spent on maintenance
- net benefits associated with maintenance performed

These statistics are essential to provide evidence to more senior management of the maintenance group's activities. Swanson further lists useful metrics by which software maintenance may be evaluated:

- average number of processing failures per program run
- total person-hours spent in each maintenance category

- average number of program changes made per program, per language per maintenance type
- average number of person-hours spent per source statement added or deleted due to maintenance
- average person-hours spent per language
- average turn-around time for a change request
- percentage of maintenance requests by type

Swanson's work is expressed principally in term of source code. The ideas can be appropriately updated and extended to address for example design documentation, requirements, specifications etc.

3.3.4 Tools for maintenance management

1. **Tools for managing maintenance**

   Gamalel-Din and Osterweil [85] report that maintenance management activities can be divided into two classes: product - related and process - related. The former is far better supported by computer aids, such as version control systems (RCS [261], SCCS [211] ...), reuse support systems (Draco [177]) and configuration control systems (NuMil [176], Odin [48] ...). The latter [163, 166, 187] which includes personnel and resource management, walk-throughs, quality audit and planning are generally done manually.

   To help maintenance managers overcome the difficulties in planning and scheduling of maintenance activities, a system has been developed at the University of Durham. SCIMM (Software Change Information for Maintenance Management) [60] stores information about requests for changes and changes made to software systems, with a view to easy access and retrieval of data, and the provision of analysis to aid managers in prediction and planning. The picture of maintenance management is based on an increase in the visibility and understanding of the work being undertaken. This takes the form of the production of progress reports, statistics and more information on individual change; this information once collected can be stored. This store of information provides a history of the work done, allowing it to be analysed to find shortcomings in the maintenance methods being employed and the process involved, so providing information to help future work.

In a more immediate sense, collection of data about the maintenance of a system increases the visibility of the maintenance process and gives information to the senior management. A record of changes to the system and their reasons, forms a permanent store of experience gained by programmers while maintaining the system.

In a study conducted by Collofello and Buck [54] it was seen that more than 50% of errors or faults were introduced by previous changes, so the *ripple effect* is a major contributor to error reports. Ripple effect [276, 277, 278] is the phenomena by which changes to one program area have tendencies to be felt in other program areas. Using the records of past changes, the original cause of ripple effect errors can be established, allowing the redesign of the original change or at least a better understanding of the problem's cause.

2. **Tools for product management**

As mention in Appendix A, three tools are commercially available for managing maintenance products: SABLIME [115] is a comprehensive product administration system that tracks changes to a product consisting of software, hardware, firmware, and/or documents, from its origination, through maintenance, delivery, and support. Its integrated Modification Request (MR) and Configuration Management capabilities make it a unique tool for managers and product developers alike [115].

RA-METRICS [251] and SMR [115] are software metric repository. RA-METRICS supports all of the management reporting metrics and it reports: functional and technical quality, user satisfaction, defects counts, CASE/Tool Usage, development and maintenance history, financial history and estimation accuracy whereas the Software Metric Repository (SMR) is a menu and mouse driven database featuring a 'point and Shot' user friendly interface. The database incorporates the software metrics generated by PC-Metric as well as Functions Points and project data. The browse and reporting capabilities are encouraging to examine and analyse the raw data.

PC-Metric is a software metric generation package. It analyses the source code and computes numerous size and complexity metrics.

A Problem Monitoring System (PMS) has been reported in [93] which has been developed for the specific purpose of controlling customer queries and problems. The PMS is a PC-based system that holds customer and product information and is driven by a series of events that are date and time stamped. PMS includes a flexible and powerful reporting facility that provides

- Statistics
  - time taken to resolve a change
  - programs most affected
  - requests outstanding
- Reports
  - list by requests
  - list by customers

3.3.5 Organising the maintenance department

The selection of the appropriate staff for a maintenance project is as important as the techniques and approaches employed. There is a remaining question in the organisation of the maintenance activity on whether or not the maintenance staff should be organised as a separate department.

1. **Development/ Maintenance department**

   The industries surveyed in [151] by E. Swanson claimed that programmer productivity was increased through programmer specialisation and that control of cost and effort was improved when maintenance activities were separated from new development. (for small organisations, maintenance may be a separate group, but a specialist may handle the maintenance work).

   Three cases studies have been reported by Swanson and Beath [257] about what kind of departmentalisation in a company is necessary with its strength and weaknesses:

   - Departmentalisation by work type (system analysis versus programming)
     - Stength: development and specialisation of programming knowledge and skill
     - Weakness: cost of coordination between systems analysis and programmers.
   - Departmentalisation by application domain (group A versus group B):
     - Stength: development and specialisation of application knowledge
     - Weakness: cost of coordination and integration among application groups
   - Departmentalisation by life-cycle phase: (development versus maintenance)
     - Stength: development and specialisation of service orientation and maintenance skills

- Weakness: cost of coordination between development and maintenance units

Many managers have indicated that separated department can improve the effectiveness of the development and maintenance ones. However, the reality of size, organisation, budget, and staff often prevent the establishment of separate maintenance and development departments.

2. **Maintenance staff**

   The maintenance staff must effectively meet the challenge of maintaining a software system while keeping the user satisfied, costs down, and the system operating efficiently. In this section, the skill level needed for maintenance, the profile of the maintainers and the organisation of the maintenance team are analysed.

   - **The skill level needed for software maintenance**

     The maintainer should be:

     - Senior
     - Experienced
     - Knowledgeable about the existing system before attempting to change it

     The maintainer has to [178]:

     - Perform all the activities of the life-cycle
     - Analyse the problem and the impacts on the program
     - Determine the requirements and design changes necessary for the solution
     - Test the solution until the desired results are obtained
     - Release the revised software to operation or the users

     Thus, there is a broad consensus that the successful maintainer needs to 'know everything about everything'.

   - **The profile of the maintainer**

     The ideal maintainer requires the following qualities [90, 178]:

     - **Flexibility:** the software maintainer must be able to adapt to different styles of code, priorities, and user requests.
     - **Responsibility:** performance of assigned tasks in a dependable, timely manner.
     - **Creativity:** the ability to apply innovative and novel ideas which result in practical solutions. Within the constraints of what is, please create something new.

- **Self-motivation:** the ability to independently initiate and complete work after receiving an assignment.
- **Discipline:** the ability to be consistent in the performance of duties and not prone to trying hazard approaches.
- **Patience:** in liaison with customer
- **Humility:** in front of criticism
- **Experience:** software maintainer should have a broader education than software developers; greater skill are needed because maintainers not have only to look forward to new techniques, but also backward to previous ones.
- **Analytic:** a maintenance programmer must be able to analyse and understand system's requirement, design, capabilities and limitations and problems, and to correct problems and add capabilities.

3. **Organising the maintenance team**

In order to accomplish the task of maintenance, there is a framework within which the maintainer has to operate consisting of user requirements, the existing program, the tools available, the environment and the maintainer's own capabilities. Maintenance staff should be organised in a manner that results in efficient use of human resources and effective application of available skills.

Usually, maintenance staff has no knowledge on the process that created the product (see section 2.4.9). So, software maintenance may be undertaken by the original development group, or responsibility may be given to a separate group. In the former case, there is a strong case to make the software maintainable, although documentation quality may suffer as staff see (even) less need for it. In the latter case, the development team may be able to move to new projects, and each group develops expertise.

The organisation of the maintenance team can be as follow:

The current practice (see section 2.4.2) reports that the maintenance activity is used as training for new employees. The Guidance on Software Maintenance from the NBS [178] suggest that 'Maintenance should NOT be used as a training ground where junior staff are left to *sink-or-swim*'. Many authors are agreed that the maintenance teams should include a mix of experienced and junior staff.

Boehm [34] suggests that the maintenance staff should be involved earlier in the software process during standards preparation reviews and test preparation.

Fairley [82] recommands that the minimum size for a maintenance team is two because maintenance personnel can work more effectively when they check one another's work and when they can learn from one another. By working with others, they are likely to find more errors sooner and to perform the job at less cost.

In larger teams, an identifiable quality assurance group should be established. There are strong advantages if this group has a separate reporting structure, to maximise its independence. The maintenance staff should be responsible, mobile and productive.

- **Responsibility:** Many authors consider that task responsibilities should be clearly defined and technical specialisation among maintainers allowed to provide a higher motivation. Each team is responsible for the maintenance of one or more software systems. A maintenance leader is defined in each teams which is directly responsible for technical program support. He reports to the maintenance manager and the rest of the team reports to him.
- **Mobility:** McClure [166] suggested the maintenance personnel should be rotated between design and maintenance. It seems appropriate to rotated between projects, to avoid individuals regarding the system as their private domain (this has management advantage also, if the person leaves). Boehm [32] suggest that someone from maintenance be a part of the Quality Assurance team during development in order to make the transition to maintenance more satisfactory. This suggestion will depend on the organisation of the company.
- **Productivity:** Boehm [10] defined software maintenance productivity as the ratio of the number of source instructions added or modified to the number of man-months of maintenance effort. A company can increase maintenance productivity by using the best method to perform this activity, providing the right procedures and tools and investing in technology, so that maintainers will have similar facilities to developers.

4. **Communication in the maintenance department**

   A good communication inside and outside the maintenance team is essential to the success of this activity.

   - **Communication inside the maintenance team**

Much of the communication is social chit-chat [135] but during these talks a surprising amount of technical communication and learning takes place during the typical coffee break.

In a maintenance environment, ensuring good communications is essential in order to prevent chaos. The maintenance communications vehicles are meeting minutes. At weekly team meeting, all members of the team are present, and each member reports on progress since the last meeting. Each team member may introduce to the group new ideas, requests for changes, and any other information he feels is to the benefit to the group. It is essential that information about both the software being maintained and the maintenance process is collected and reported to management.

To improve communication, we must encourage electronics mail which is fast, less expensive, more often read and answered.

- **Communication between maintenance team manager and upper management**

  A major problem faced by the maintenance community is the lack of recognition by senior management of the problems of maintenance and the benefits and returns on investment that maintenance work provides. The reason behind this is the lack of effective communication between managers of the maintenance team and the senior management responsible for running the business.

- **Communication between maintenance team and user**

  A user is a person or group that is directly involved in the actual use of the system. It is very important to provide good training for users. If they understand well the system, they will require less from the maintenance staff. The maintenance team works with users and try to understand the problem as expressed in the user's language. Then, the problem is transformed into a request modification. The change request includes a description of how the system works now, how the user wants the system to work and what modifications are needed to produce the change. The user interface can often be established with the concept of 'hot-line' or 'help-desk' (see section 3.4.2) .

3.3.6 Managing the maintenance team

Talented people comprise the single most important element of a software maintenance organisation. The crucial initial step is to make available the best people from within the organisation to work in a well-structured and well-managed environment in which they can function as a team. Well-motivated staff are likely to ensure stability within the maintenance organisation, adequate training is vital to ensure that the maintenance team is highly productive.

1. **Motivating the maintenance staff**

   Some organisations have tried to improve the maintainer's motivation and the image of the maintenance activity by simply giving another name which is a superficial approach. It changes nothing except the name. A better approach is to acknowledge the importance and value of good maintenance to the organisation through career opportunities, recognition, and compensation.

   Here is some advices given to obtain a high motivation in the maintenance team [34]:

   - Couple software maintenance objectives with organisational goal and link its rewards to organisational performance.
   - Integrate software maintenance personnel into operational team.
   - Create a discretionary corrective maintenance budget.
   - Rectify the negative image of software maintenance. If a company wants to hire an engineer for the software maintenance team, it should talk about 'optimisation of existing systems' instead of software maintenance.
   - Clearly showing the importance of maintenance to organisational goals.
   - Monitor the staff turnover which vary from location to location:

     A rate under 10% is considered normal but much higher rates may be seen as a big problem and senior management should probe to determine the cause [199]:

     – Staff compensation adequate.

     – Working environment adequate to motivate staff

     – Clear career structure: have advancement paths been established (position advancement are important to motivate software maintenance staff with promotion criterias, frequencies ...).

   In a nutshell, management have to demonstrate that maintenance is of equal value and is as challenging as software development.

2. **Training**

Methods, tools and environments are changing so rapidly that it is difficult to determine what kind of training does the maintenance team need.

A Brief look at existing training has to be analysed by:

- Basis: conceptual, procedural, survey, system specific, tool specific
- Level: strategic, managerial, technical
- Form: university, in-house, public, part of other courses, informal, on the job

Obstacles for sound training in software maintenance remain but opportunities are given to cope with this lack:

- Obstacles: lack of accepted model and theory, inappropriate emphasis on development, lack of interest
- Opportunities: university level courses with software engineering analysis and management, and introduction of methodology with each new technique or tool and team by team.

There are some existing courses on software maintenance:

- The University of Durham and the Polytechnic of Liverpool both include software maintenance issues within their software engineering courses.
- The University of Durham has begun trials of courses for industry aimed so far at management.
- Brown University (USA) includes at least some software maintenance issues in its Computer Science course
- Richard Ball (from Canada), Bob Wachtel (from the USA) and Nicholas Zvegintzov (from the USA) give seminars on the subject, with occasional appearances in the UK.
- Integrated Computer Systems (ICS), a commercial training organisation, has introduced a three-day course to run three times a year in UK.

A maintenance training should be composed with courses on the application domain, software engineering, and syntax knowledge.

In general, a company should be spending between 2% and 5% of software budget on training that is between 7 and 15 days per year [199].

A good training plan provided to the maintenance team is important to its motivation and to adequately perform its work.

3.3.7 Conclusion

In this section a good management of the software maintenance activity and the software maintenance process is given along with advices, the type of data to collect and the tools to use. A maintenance plan should be provided for the management of the maintenance activity. A software configuration management tools should be used for controlling and monitoring the software maintenance process. The maintenance department should provide a good training plan for the maintenance staff and increase the staff motivation. Some data on each project should be collected and statistics provided to senior management.

After defining the management of the maintenance activity, there is a need to understand the different tasks involved in the software maintenance process and the technical information required to perform maintenance.

3.4 The Technical side of Maintenance

3.4.1 Introduction

This section surveys the different software maintenance tasks models, presents an adequate maintenance process for Aerospace systems and provides technical information required to achieve maintenance.

3.4.2 Software Maintenance Models

Modelling the software process is an important current area for research [262]. This is not surprising, since in order to understand and assess a software *product*, we need to understand and study the *process* by which it was produced. Much of the work is addressing the initial software development, although some research is also including the evolution and more general maintenance of software.

There would appear to be two approaches to work on software process models. In the first, the *descriptive* approach, existing software development is observed in real life, and empirical conclusions are abstracted from such analysis. In contrast, process models may be *prescriptive*, so that a model derived from theoretical or abstract considerations is imposed on the software development process. In practice, research is likely to move forward by a combination of these approaches, and this reflects the description of maintenance models given in this chapter.

Current Models

The traditional life cycle model of software has always shown the software maintenance activity as a single step at the end of the cycle. This model is summarised below (for more details see section 2.3.3):

1. Requirements
2. Design
3. Implementation

4. Test
5. Installation and check-out
6. Operation and Maintenance

with possible feedback loops from each phase.

The model does not portray the system life; it only shows the creation and development (or youth) of a system. It does not show the evolutionary development (or adulthood) that is the characteristic of most software systems. The final step needs to be replaced by a model that reflects this aspect of software evolution.

A number of authors have proposed models of the software maintenance process [154, 31, 234, 185, 163, 276, 189, 181, 14, 197]. The following is a summary of software maintenance tasks models reported in the literature:

Boehm [31] outlines three major phases of a maintenance effort in his model:

1. understanding the existing software
2. modification of the existing software
3. revalidation of the modified software

The Martin-McClure model [163] is similar, consisting of three tasks:

1. program understanding
2. program modification
3. program revalidation

Parikh [185] has formulated a description of maintenance tasks that offers a very complete step by step protocol which may be followed for a maintenance assignment:

1. identification of objectives

2. understanding the software
3. modification of the code
4. validation of the modified program

Sharpley [234] highlights more directly the process of correcting errors in existing systems:

1. problem verification
2. problem diagnosis
3. reprogramming
4. baseline verification/reverification

Yau [276] focuses his model on software stability through analysis of the ripple effect of software changes. The five major activities of this model are:

1. determining the maintenance objective
2. understanding the program
3. generating a maintenance change proposal
4. accounting for the ripple-effect
5. regression testing the program

The Patkow model [189] concentrates on the front-end maintenance activities of specifying the maintenance requirements. It consists of five generalised steps:

1. identifications and specification of the maintenance requirements
2. diagnosis and change localisation
3. design of the modification
4. implementation of the modification

5. validation of the new system

The two first steps depend on the software maintenance activities (either perfective, adaptive or corrective).

Osborne [181] identified a model with more comprehensive phases:

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 their debugging
9. review of proposed code changes
10. testing
11. update documentation
12. standards audit
13. installation
14. user acceptance
15. post installation review of changes
16. completion of task

Osborne points out that although the processes are presented in a linear fashion there are a number of iterative steps involved within the model itself. For example, the results of the design review may necessitate additional design analysis or even modification to the original change request. Rapid prototyping can easily be applied to such models.

The maintenance models described here do not incorporates metrics explicitly as a method for assessing and controlling change. The use of software metrics has been successfully applied to the problem of software maintenance [123]. Methods based on metrics can facilitate maintenance tasks, improve the quality of the results and predict the need for further maintenance effort [266, 147]. Rombach and Ulery [216] propose a method of software maintenance improvement by focusing the goal, question and specific measurements associated with activities in the context of a software maintenance organisation. The paradigm (goal/question/metrics) is based on the principle that effective measurement procedures should be derived with a top-down approach from goal. It suggest that the measurement needs to start with a precise specification of the goals, continue with the refinement of each goal into a set of quantifiable questions, and end with a derivation of a set of metrics. However, their method do not specify a framework for metrics that supports impact analysis in the software maintenance process.

Pfleeger [197] describes a model for the software maintenance process that depict where and how metrics can be used to manage maintenance. The management of maintenance controls the sequence of the different activities (with a number of iterative steps) by receiving feedback with metrics and determining the next appropriate activity. The major activities are:

1. change request
2. analyse software change impact (impact/scope, traceability)
3. understand software under change (complexity, volume, documentation)
4. implement maintenance change (adaptability)
5. account for ripple effect (stability)
6. retest affected software (testability, verifiability)

The analysis and monitoring of the impact of change coupled with feedback and metrics, allows management to confirm if the change meets the requirements, does not degrade the maintainability and is being implemented in the best way.

Request-Driven Model

Elements from these models can by combined to produce a task model that describes in detail the activities that take place during maintenance. This model is a **Request Driven Model** that attempts to portray the activities of software maintenance as dictated by users' requests for change. The model consists of three major processes [23] called :

1. Request Control
2. Change Control
3. Release Control

It should be noted that the word *Control* has been deliberately used at the end of each process name to imply that the model will not work effectively without strict control from management of all the activities that take place.

The activities that take place in each of these process are now described :

1. Request Control

   The major activities are :

   - collect information about each request
   - set up mechanisms to categorise requests
   - use impact analysis to evaluate each request in terms of cost/benefit
   - assign a priority to each request

   This initial step of collection should be carried out by a ' Help Desk' manned by staff who will not be directly involved with the technical process of satisfying the request. It is preferable if the Help Desk is staffed by highly skilled Systems Analysts; such people can also distinguish genuine user change requests from queries arising from the misunderstanding of user documentation etc.

2. Change Control

   The activities during the change control process are :

- o select from top of priority list
- o reproduce the problem (if there is one)
- o analysis of code, documentation and specifications
- o design changes and tests
- o quality assurance

3. Release Control

   The activities are :

   - o release determination
   - o build a new release
     - edit source
     - archival and configuration management
     - quality assurance
   - o confidence testing
   - o distribution
   - o acceptance testing

3.4.3 Software Maintenance Model for Aerospace Systems

The different task models outlined in the previous section do not reflect the requirements for Aerospace systems because their products usually have only one customer, they use higher level languages like Ada, C and Fortran instead of Cobol and the staff is more skilled than for business application; they sometimes need to do some quick fix for maintenance because of the criticity of systems and they apply Aerospace system standards (and also see section 5.3).

The models from Liu, Boehm and Osborne [154, 31, 181] do not give any details of the different tasks. Sharpley's task model [234] is only dedicated to corrective maintenance and there is no information on retesting. Parikh, Yau, Pfleeger [185, 276, 197] are general task models.

Patkow's model [189] is very interesting because of its generalised model and its refined versions for the different maintenance activities, but does not mention the need to check for the effects of a change on those portions of a program that were not actually modified.

The Request-driven model [20, 23] is dedicated to companies with many customers and for business applications.

Thus, for Aerospace systems, the following general model for software maintenance tasks is proposed:

1. Identification of need for modification
2. Program comprehension and localisation of modification
3. Design of the modification and impact analysis
4. Implementation of modification
5. Revalidation

with software configuration management and quality assurance

This general task model can be refined for the different software maintenance activities: corrective, perfective, adaptive and preventive.

1. **Identification of need for modification**

   - Corrective Maintenance

     This activity starts when anomalous behaviour is observed to have occurred within the system and as a result an anomaly report is issued by the operations staff. The necessary maintenance action is then carried out by the maintenance staff [242]. The current request with the previous maintenance requests is compared in order to determine similarities. The task of identification is to reproduce fault situations, verify reported problems and specify correct operation of the system. This task requires test data and environment simulator (see section 6.9.3).

   - Perfective Maintenance

     Identification of a deficiency in functionality and specification of desired functionality. Identification of new or altered requirements and the specification of the operation of evolutive system. A modification request is issued.

o Adaptive Maintenance

Identification of a change in the processing or data environment, describe the change and the revised specifications to reflect it. A problem report is issued.

o Preventive Maintenance

Identification of deficiency in maintainability or quality standard and specification of the desired quality standard. A problem report is issued.

All Anomaly reports, problem reports or modification request are submitted to the Configuration Control Board (CCB) consisting of representatives from the maintenance team, quality, configuration management and project management teams. The CCB has to approve on a reported problem. If approved, a Software Change Request(SCR) is issued internally.

2. **Program comprehension and localisation of modification**

o Corrective Maintenance

Localisation of the part of the system which is responsible for the error. The diagnosis of errors in a large system is often the most difficult and time consuming task in corrective maintenance [234]. An anomaly diagnosis is carried out to identify the failed component or module. Then, the failed items are identified, an analysis of the problem is carried out to determine the cause of the failure and the form of corrective modification required.

Maintenance tools:

- code analyser (see section 4.2.1.-1 and A.1.1)
- code visualisation (see section 4.2.1.-2 and A.1.2)
- debugger (see section 4.2.1.-5 and A.1.5)
- cross referencer (see section 4.2.1.-3 and A.1.3).

o Perfective Maintenance

Localisation of the source of the deficiency or of the existing software elements which are affected by the new requirements (requirements, specification, design, code, test data). The maintainer has to find where the resources were excessively consumed in order to make an optimisation.

Maintenance tools:

- code analyser (see section 4.2.1.-1 and A.1.1)
- code visualisation (see section 4.2.1.-2 and A.1.2)

  - debugger (see section 4.2.1.-5 and A.1.5)
  - cross referencer (see section 4.2.1.-3 and A.1.3)

- Adaptive Maintenance

  Localisation of all software elements affected by the change. When there is a change in the data environment, the maintainer must find the parts of the system that use or set the data that is being changed. It is necessary to have some knowledge of what the system inputs and outputs are, where they are used, and what their properties are. This knowledge can be kept in a data dictionary.

  Maintenance tools:

  - code analyser (see section 4.2.1.-1 and A.1.1)
  - code visualisation (see section 4.2.1.-2 and A.1.2)
  - debugger (see section 4.2.1.-5 and A.1.5)
  - cross referencer (see section 4.2.1.-3 and A.1.3).

- Preventive Maintenance

  Localisation of the existing elements which are concerned with the modification request. The maintainer has to to locate the part of the software where there is a lack of maintainability.

  Maintenance tools:

  - code analyser, quality analyser (see section 4.2.1.-1 and A.1.1)
  - code visualisation (see section 4.2.1.-2 and A.1.2)
  - debugger (see section 4.2.1.-5 and A.1.5)
  - cross referencer (see section 4.2.1.-3 and A.1.3)

For more details on program comprehension and fault localisation see also section 4.3.1. and 4.3.2.

3. **Design of the modification and impact analysis**

This task decides what the correct properties should be, how these properties are to be established, and determines the extend of any ripple-effect.

The impact analysis evaluates the effects of a proposed change. This activity determines whether the change can be made without adversely affecting the rest of the software. Determining the impact of the change is an evaluation of the number and size of system artifacts that will be affected by the change. Traceability suggests the connectivity of the relevant workproducts and whether traceability can be established once the proposed change is made. If the impact is too large, or if the traceability is severely hampered by the change, management may choose at this point not to implement the change.

The design of modification requires an examination of the side-effects of changes. Dependencies are easier to establish when properties are stated explicitly, and are traceable in the design and code. Components can be troublesome if they are complex, or highly coupled to other parts of the system.

In this phase, the problem severity is evaluated, a proposed modification is carried out on a feasibility study, an estimation of the modification cost made and the side-effects minimized.

This involves a search through the specification, design, code, test suites, documentation ... within the changed module and continuing to all other modules which share global variables or common routines with the changed module. Also, only one module should be changed at a time and the potential ripple effects of each change should be determined before changing the next module in sequence.

Maintenance tools:

- test coverage monitor (see section 4.2.3.-1 and A.3.1)
- test regression testing (see section 4.2.3.-2 and A.3.2)
- test impact analysis (see section 4.3.3. and B.3
- test ripple effect analyser (see section 4.3.3.-2 and B.3.2)
- test modification cost

This phase ends with acceptance/ rejection of the proposed modification by the CCB.

4. **Implementation of Modification**

   Implementation of all the modifications identified in the impact analysis phase. The maintenance tools that can be used are the same as the previous phase, but for preventive maintenance we need to use reverse engineering tools (see section 4.2.2. and A.2).

5. **Revalidation**

   This phase is to ensure the reliability of the modified system. The process of revalidating a program [163] consists of system testing, regression testing (test of unmodified portions), and change testing (test of modified portions). The original test cases and test data should be utilised as much as possible.

- Change testing: testing the modified portions of the program to determine if the change was designed and implemented correctly.
- Regression testing: testing the unmodified portions of the program to determine if those areas still operate correctly.
- System testing: testing to be ascertain that the entire system as a whole is still operating correctly.

Furthermore, for perfective maintenance, there is a need to compare the performance before and after modification or to test the validation of the new or altered requirements.
Maintenance tools:

- test coverage analyser (see section 4.2.3.-2 and A.3.1)
- regression testing (see section 4.2.3.-1 and A.3.2)

The software maintenance process ends after the user has accepted the modified system and all documentation has been satisfactorily updated.

If a severe error exists (e.g. a critical system cannot function), maintenance staff are immediately assigned to the software maintenance process and some of the maintenance tasks can be shortened.

Other Activities

Other activities are performed in parallel with the software maintenance process:

1. **Software configuration management**
   The preliminary objective of the software configuration management [181], generally referred to as the management of software modification, is the release of operationally-correct, and cost-effective software. SCM is an integrated set of four subdisciplines:

   (a) Software Configuration Identification is the definition of the different baselines and associated components of a system, and any change made to these components and baselines.

   (b) Software Configuration Control is the control procedures for making changes to components and baselines.

(c) Software Configuration Status Accounting is the provision of an administrative history of the evolution of a software system.

(d) Software Configuration Audit determines whether or not baselines meet their requirements.

SCM helps to ensure that software change satisfies specified requirements and change criteria.

SCM also should provide record retention, disaster recovery, library activities, a software repository, and ensures that the necessary coordination and approval are obtained prior to changing the baseline. SCM helps to track all actions associated with problem reports or change requests.

Maintenance tool: Software configuration management (see section 4.2.4. and A.4.1).

2. **Quality Assurance**

   Software quality assurance is an activity that is applied throughout the software life cycle. It encompasses [198]:

   - analysis, design, coding, and testing methods and tools.
   - formal technical reviews that are applied during each phase of the software life-cycle.
   - control of software documentation and the changes made to it
   - a procedure to assure compliance with software development and maintenance standards (when applicable).
   - measurement and reporting mechanism.

   It is essential that all change considered for a system are formally requested in writing. The analysis of the complexity of the software and the relationship between products is essential to determine whether the overall maintainability of the system will enhanced or degraded by the modification.

   If the Configuration Control Board is unhappy with the likely degradation of these system characteristics, the desirability for the change may be reassessed or the way in which the change is to be implemented may be reevaluated.

   The maintainability measures give management and customer an idea of the likely overall quality of the resulting product. By monitoring product quality with each change, the software maintenance process model can be used to increase overall quality and enhance maintenance productivity.

The most formal maintenance review occurs at the conclusion of revalidation and is called configuration review. The configuration review ensure that all elements of the software configuration are complete, understandable, and filed for change control. During development, maintainability reviews should be conducted repeatedly as each step in the software engineering process is completed. Maintenance tools: quality analyser (see section 4.2.1.-1 and A.1.1).

3. **Maintenance repository**

   It is essential that a maintenance repository that store information associated with each project throughout the project life cycle is established and preferably computer-based that will contain:

   - all change requests
   - progress of change
   - modification impact list
   - list of known errors and omissions
   - test database
   - hardware and software history
   - review reports
   - system configuration data
   - revalidation data
   - reliability data

   Maintenance Tool: Product Management (see section 4.2.4. and A.4.7).

3.4.4 Technical Information for Maintenance Staff

Information for maintenance can be collected from a number of sources, including the source code, internal documentation, external documentation, the system developers, and any body else who happens to know about the system.

The previous section has described the maintenance tasks through different software maintenance models and this leads to identify the technical information necessary to perform effectively software maintenance [189, 53]:

1. **Requirements information:**

   It is important that both the functional and performance requirements for the existing system be known and understood so that they can be preserved during the modification process. A sufficient understanding of the application area should be provided which includes the perceived needs and desires of the end users.

   The additional requirements planned or anticipated but not implemented should be identified.

2. **Specification information:**

   Maintenance staff need to have knowledge of both high and low level system behaviour which includes what a system does and how it does it and why the user wants the system to do this.

   The functional specifications should be organised in a manner that they can explain the relationship and interactions between functions.

   The method used for the specification phase and the way to modify it should be explained.

3. **Architectural and low level design information:**

   Design principles and decisions should be understood. This includes why certain design alternative were chosen and others were disregarded. Maintenance personnel need to have knowledge of all algorithms that are used regardless of their complexity. Coding style, standards and other implementation convention should be understood.

4. **Processing environment information:**

   The interactions within the total processing environment need to be known. This includes resource requirements such as the hardware and support software.

5. **Declaration, control and data flow information:**

   At the source code level, a maintainer must have knowledge of a program's control flow, its invocation hierarchy, data flow, data aliasing, loop termination conditions, entry and exit assertions for all procedures (functions), and all over syntactic elements which contributes to an understanding of a program's run time behaviour. A maintainer must also have knowledge of the declaration information for all data object in a program.

6. **Traceability between life-cycle products:**

   Knowledge of the functional and performance requirements of a system should ideally allow a maintainer to relate what the system does with the various software components that causes it to function.

This implies traceability from the specification to the design and source code. Furthermore, knowledge of software requirements should contain an understanding of the relationship and interaction between different software functions.

7. **Test environment information:**

   Comprehensive knowledge of diagnostic and regression tests must be available as well as an understanding of how to use them. This will include available test cases, expected results and a test case history.

8. **Document organisation information:**

   The maintainer should also understand how the specification, design, test cases, etc are organised, this will allow quick localisation of items of interest rather than unnecessary lengthy searches through documentation. It is important that the maintainer has access to the appropriate level of detail at any point in time. If there is a document dedicated to maintenance, it should be stored in a database based on hypertext technology in order easily to browse in the documentation.

9. **Anticipated features for enhancement:**

   The maintainer should have knowledge of anticipated additional features that have not yet been implemented, This would avoid redundant effort spent in rethinking requirements analysis and design.

3.4.5 Conclusion

In this section, software maintenance tasks model have been surveyed and a model for software maintenance of Aerospace systems proposed with useful proposed tools at the different phases. This model is described with the different tasks e.g. identification of need for modification, program comprehension and localisation of modification, design of the modification and impact analysis, implementation of modification and revalidation along with software configuration and quality assurance. The use of software metrics applied to maintenance has been emphasised to facilitate maintenance tasks, improve the quality of the results and predict the need for further efforts. The technical information to carry out maintenance has been presented.

3.5 Summary

This chapter has presented a report on software maintenance best practice based on an analytical approach of the software maintenance process and addressed at three different levels: organisational, managerial and technical in order to define the best method for maintenance. The senior management should be involved to define the maintenance strategy according to the overall one of the company. The maintenance activity should be seen as a product support organisation. The improved software maintenance process should be planned, monitored and controlled with appropriate tools. The maintenance department should be managed in an efficient manner in order to increase its efficiency, productivity and the motivation of the maintenance staff. A model for the software maintenance tasks dedicated to Aerospace systems has been presented and the importance of data collection and software metrics emphasised.

Now, we shall define and survey the maintenance tools available to perform maintenance and the on-going research projects and prototypes.

Chapter 4

Software Maintenance Tools

4.1 Introduction

Several surveys on software maintenance tools have been reported. The General Service Administration's Office of Software Development and Information Technology (GSA) devised software maintenance tools into eleven categories and brought them together in what is termed a Programmer's Workbench(PWB) [243]. The PWB is specifically oriented to Cobol applications on IBM architectures. The Software Maintenance News surveyed Software Maintenance Tools in 1989 [244]. These are dedicated both to scientific and business applications.

The objective of this chapter is to present a survey on commercially available tools and prototypes that can be useful for the software maintenance process. The scope of the survey is to discuss software tools which can be used in the maintenance of scientific or real-time systems. This survey is not exhaustive but the tools listed are meant to be representative of the techniques that are currently commercially available. With the prototypes and research projects in progress, it will be easier to outline the research trends in order to make better and more efficient tools.

The information on tools was obtained in our survey from several sources including product descriptions from tool vendors, earlier survey articles [273, 180, 251, 243, 244], conference papers [111, 112, 113, 114, 115], and research papers.

This chapter is divided into two parts:

1. Commercially available tools
2. Prototypes and research projects

Further comprehensive details of tools and prototypes are given in Appendix A and B.

4.2 Commercially Available Tools

In this section, commercial tools are classified into four categories: tools that are useful for program comprehension, reverse engineering, testing and maintenance management.

4.2.1 Tools for Program Comprehension

Program comprehension is the most expensive phase of the software life cycle [246] and it is suggested that 40 % of the maintenance effort is spent in trying to understand how the existing software works. It seems worthwhile then to investigate tools and techniques to reduce these costs and then a significant saving can be made.

Code Analysis is used for examining a piece of program code, and is used for determining the dependencies between different entities and analysing the usage of entities. Different techniques are used:

- data flow analysis examines the piece of program code in order to determine if there are any anomalous use of variables within that code.
- program slicing is a form of program decomposition based on control flow and data flow analysis in coherent modules.
- call graphs is a directed graph that represents the dynamic relations between routines and calls. This technique make use of control flow graphs in order to perform the desired analysis.

- program transformation systems systems are systems that transform a program into a program structurally different but logically equivalent. Program transformation tools are divided into restructurers and formal transformations.

Many tools can be useful in supporting program comprehension at the code level e.g. code analyser, code visualisation, cross referencer, source code comparer and execution monitor/ debugger. We shall now investigate these in more details.

1. **Static Code analyser**

   Static analysers include tools that analyse a program's control structure data flow. By static we mean that the program itself is not executed and therefore its run-time behaviour is not executed.

   VAX SCA, ISAS and F-SCAN are common static code analysers (for further details see Appendix A):

   (a) VAX SCA provides facilities such as logic tracing, data flow tracing, and consistency analysis as well as a cross referencer

   (b) ISAS reports and charts procedure hierarchy, data references, control flow, system structure

   (c) F-SCAN provides structure charts, Call/Vcalled tables, Set/Used tables, and diagrams of Common.

   (d) MALPAS [250] provides control flow, data use, information flow, partial programmer, semantic and compliance Analyser.

   ACT, BATTLE MAP and LOGISCOPE also calculate software complexity metrics that are useful for quality assurance.

   (a) ACT [165] is driven by and analyses source code, producing a graphical representation of module structure, and also calculates the McCabe cyclomatic complexity metric and generates the basis set of test paths that should be exercised for each module within the source code.

   (b) BATTLE MAP [165] allows the user productively to reverse engineer large existing systems by providing a comprehensive, visual understanding of the entire program structure along with its quality attributes .

(c) LOGISCOPE [169] shows the internal logic structure of each module of code, as well as the structural relationships of all the modules.

The results provided by the Complexity Analyser are:

- quantitative with Halstead, McCabe and Mohanty metrics;
- qualitative with control graphs, call graphs, criteria graphs and Kiviat diagrams;

LOGISCOPE is a very useful tool for Quality Assurance with the software complexity analyser.

2. **Code visualisation**

Code visualisation[180] is a tool to help the maintainers analyse and understand the code through a powerful man machine interface. While all these tools show how a program is structured, they use different means to achieve different ends.

(a) OBJECTIVE-C Browser [180] uses a windowing approach that displays hierarchical and functional information about code object in C or Objective-C.

It provides three types of information about the source code:

- the contents
- available cross-referencing data
- source code contents with respect to the inheritance hierarchy.

(b) VIFOR [202] on the other hand, takes a different approach for FORTRAN visualisation. The browser has a graphical interface that allows the user to select, move, and zoom into icons representing parts of the program. As such, it gives a graphical editing capability.

(c) SEELA [98] takes yet another approach to visualisation by using Reverse Engineering. It converts code into a program design language and lets the user edit the structure chart, cut and paste to and from the code, and generate high-level documentation describing the code structure. Thus, it gives and logical path between code and its corresponding design language.

Unlike SEELA, which works with many languages,

(d) GRASP/ADA [180] is an example of a comprehension tool tailored to specific language. This tool builds graphical control-structure diagrams that high-light the control paths in and among Ada tasks.

(e) ACT and BATTLE MAP [165] take an entirely different approach to visualisation control path in graphs of all control paths in order easily to produce the control flow and corresponding complexity.

(f) EDSA [265] uses program slicing for collapsing and depicting large amounts of code in a small window.

3. **Cross referencer**

   These tools trace the use of data elements, named paragraphs, and/or procedures through a program. Object references are usually identified by source statement numbers. Associated with the statement numbers may be additional information such as the type of statement involved (move, assignment, conditional, etc) or perhaps a copy of the statement itself. These tools will identify all occurrences of data names, words or literals within a program. It is a useful tool for navigating around source code.

   Cross referencers will typically:

   - identify and trace data element modification, branch logic and program calls
   - generate graphic record layouts visually to communicate data structures and formats

   Cross referencers output is typically either a printed report or an on-line display.

   Examples of commercial cross-referencers e.g. ADPL, Autoref, BPA, CICS-OLFU ... are given in Appendix A.

   Cross-referencers are very common maintenance tools usually used with interface analysers and code analysers in order to understand the source code.

4. **Source Code Comparison**

   These tools are designed to help programmers quickly identify changes between program code versions.

   ISAS, Matchbook, S/Compare, Text Comparator detect and highlight differences between two or more files including additions, deletions and in some tools, moves. Source code comparison can be a significant aid in determining the rational for previous undocumented maintenance and should be included in version control.

5. **Execution Monitoring /Debugging**

   This group of tools allows the programmer interactively to monitor and manipulate the progress of a program as it executes. In doing so, the maintainer can directly examine the behaviour of a program and the effects of various inputs. It is important to distinguish between the terms fault localisation, fault repair, and debugging. Myers defines debugging as:

> The activity that one performs after executing a successful test case (successful in a term that it found an error).

Deścribing it in a more concrete terms, debugging is a two part process; it begins with some indication of the existence of an error (e.g. the result of a successful test case), and it is the activity of:

(a) determining the exact nature and location of the suspected error within the program.

(b) fixing or repairing the error [174].

Thus debugging entails both fault localisation and repair.

Ducassé and Emde [74] review 18 existing automated systems on program debugging and a dozen cognitive studies on debugging. A large subset of the following are related to the tutoring task (IPA [220], Pudsy [157], Laura [2], Phenarete [270], Proust [116, 118], Talus [173], Apropos [156]).

Another group is composed of general purpose debugging systems (Sniffer [232], Kraut [37], Focus [159], Falosy [228]).

All the other systems are enhanced Prolog tracers (PTP [76], Preset [258], Opium [233]). A particular group of enhanced Prolog tracers is made out of the systems following the trend established by Shapiro in Algorithmic Program Debugging [233] (RD [191], DED [155], EDS [84]).

The table No 4.1 summarises the result of their study.

The debugging knowledge types that have been identified are:

- knowledge of the indented program (intended I/O, Behavior, Implementation).
- knowledge of the actual program (intended I/O, Behavior, Implementation).
- understanding of the programming language (Lan).
- general programming expertise (Pro).

- knowledge of the application domain (Dom).
- knowledge of the errors (Bug).
- knowledge on debugging methods.

and the classification of global debugging strategies are:

(a) filtering.

  - tracing algorithms.
  - tracing scenarios.
  - path rules.
  - slicing/dicing.

(b) checking computational equivalence of intended program and actual one.

  - algorithm recognition.
  - program transformation.
  - assertions..

(c) checking the well-formedness of actual program

  - language consistency checking.
  - plan recognition.

(d) recognising stereotyped errors.

  - error cliche recognition.

Debuggers	KNOWLEDGE of										Strategies
	Intended			Actual							
	I/O	Beh	Imp	I/O	Beh	Imp	Lan	Pro	Dom	Bugs	
Focus				*		*					a
Pudsy	*				*	*	*	*			b
Phenarete						*	*	*			c
Talus	*		*			*	*	*			b
Laura			*			*		*	*		b
IPA			*			*	*			*	b+d
Proust			*			*	*	*		*	b+d
Apropos		*	*	*	*	*	*			*	a+b+c+d
Falosy		*		*	*	*	*	*	*	*	a+c+d
Sniffer		*			*	*	*	*		*	a+d
Preset					*	*	*				a+c
PTP		*			*					*	a+d
APD		*		*	*						a
EDS		*		*	*						a
DED		*		*	*						a
RD		*		*	*						a
Kraut		*			*						a
Opium					*						a

Table No 4.1: Knowledge and Strategy Summary [74]

4.2.2 Tools for Reverse Engineering

Reverse Engineering tools are used to provide better program comprehension when the code is unstructured, poorly written or documented, or when system requirements or design are unavailable.

Chikofsky and Cross [46] defined reverse engineering as:

The process of analysing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form or at a higher level of abstraction.

Different techniques are used at different level of abstraction.

1. **Restructurer**

   Restructurers accept unstructured code as input and produce a structured program with the same functionality as output. Restructuring is the transformation from one representation to another at the same level of abstraction. If a system is old, poorly documented and poorly structured, it might be possible [135] to restructure it possibly incrementaly. This can take much effort, and might be worth while only if the system is expected to be maintained for several years. Some problems with automatic restructurer have been identified by Calliss [38]. For example the amount of code produced by a restructurer is usually greater than the original program, and many of the restructuring algorithms make use of state variables which the restructurer adds to the program.

2. **Reformatter**

   Reformatters improve the format of a source file for easier reading. Tools in this category are also known as pretty-printers, and as such will layout source code in a standard format. They transform old, large, or poorly written or documented programs into standardised formats that are more readily and easily maintained. Example are given in Appendix A.

3. **Re-engineering**

   Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

   Here is an example of a set of software re-engineering tools:

   BAL/SRW [133] helps to recover the design of an assembly language program. This is achieved through a series of abstractions, which effectively collapse program functionality into progressively higher level concepts. The program is analysed and its internal representation is created in the knowledge base. In order to learn about the program logic the analyst can

   - Search for programming patterns present in the program and replace them with natural or formal language sentences in order to make the code more understandable.

- Navigate through both the source code and control flow view of the program.
- Simplify the program automatically by recognising control flow patterns, identifying subroutines and unreachable sections of the code, and by hiding selected control flow paths upon specified conditions.
- Simplify the program manually by substituting analyst defined comments for program sections.

4. **Reverse Engineering**

Reverse Engineering [273] involves the identification or 'recovery' of program requirements and/or design specifications that can aid in understanding the program to modify it.

During maintenance, the maintainer needs to develop a level of program comprehension commensurate with the task in hand. When this necessitates the recovery of lost or otherwise unavailable information concerning system requirements and/or design, it involves a complex cognitive process called Reverse Engineering.

(a) PRODOC [225] is well suited for renovating existing systems. It uses FLOWforms to represent systems at arbitrary levels of abstraction in a highly interactive visual environment. Among other things:

- the use of FLOWforms helps eliminate representational inconsistencies and awkward transitions between analysis and design.
- high level designs can be translated automatically into any of the languages supported by PRODOC.
- existing source code can be reverse-engineered.
- PRODOC can convert code from an old environment to news one with its ability automatically to translate between pseudocode languages.

(b) The Software Refinery [180] is a family of products for building automated software processing tools that take source code as input and/or produce source code as output. Software refinery includes three products: REFINE, DIALECT, and INTERSTA.

- REFINE is a programming environment for building software analysis and transformation tools. Its features a very high level executable specification language, a specification language compiler, an object oriented database, a customized editor interface, and tracing and debugging tools. This tool translates source languages into a specifications language (PSL/PSA). This tool converts (using reverse engineering) code into program design language and lets the maintainers edit the structure chart , cut and paste to and from the code, and generate high-level documentation describing the code structure. Thus, it gives an electronic path between code and its corresponding design language.

- DIALECT is a tool that generates programming language parsers and printers from grammars. It includes a high level language for specifying language, a specifying grammars and a grammar compiler.
- INTERVISTA is a toolkit for building graphical interfaces to Software Refinery applications. It provides windows, diagrams, menus, and hypertext.

Software refinery analyses and translates software in the programming language that need to be used.

4.2.3 Tools for Testing

1. **Regression Testing**

   Regression testing [252] verifies that only desired output changes occur from modified programs. Ideally, with each required change, all test cases should be re-executed and the results re-evaluated from unit level through system level testing. However, schedule and resources constraints almost always prevent this from occurring when modifying large software systems.

   Regression testing tools execute various test cases using prerecorded keystroke inputs and then compare actual results of the current test session with expected results. These tools [251] are characterised by the high level capabilities of capture, replay, and compare.

   - *Capture* is the capability of recording the input (scripts) and outputs (benchmarks) of a test session. Typically, inputs consist of keyboard inputs and outputs consist of terminal screen displays. Text editors create and modify script files or test drivers for some regression tools.
   - *Replay* is the capability of reissuing prerecorded inputs (playing back the script). This ensure that test case inputs are the same as in previous tests and minimize the tedious, error-prone procedures that must be executed.

- *Compare* is the capability of determining that the actual results of the current test session are the same as those previous test session (benchmarks). This allows the tester to focus his attention on resolving discrepancies instead of locating the discrepancies.

Different regression testing tools are reported in [252] and each of them provides one, two or three of the regression testing characteristics:

- Capture regression testing tools.
  (a) 'Bloodhound' [252] captures an unlimited number of keystrokes and screens in text mode. Screen images can be automatically captured whenever the screen scrolls. Screens can also be captured at arbitrary points in the user program. Tests can be run after changes to see if any regressions have occurred.
  (b) 'VAX/Test Manager' [252] automates the regression testing of software. TM runs user-supplied tests, and the results are automatically compared to their expected results. Regression testing assures that changes have not affected the previous execution of the software. TM operates both in interactive and batch modes. It has a DEC windows interface which is consistent with other window applications, making it easy to learn.
- Capture-replay regression testing tools.
  (a) AUTOMATOR [252] provides repetitive task automation. It supports regression testing by the recording of scrips and also the writing of scrips in a scripting language. Performance testing is provided by a function that records the screen, keyboard and internal lock, thus providing execution data. AUTOMATOR has the ability to generate random tests given an array of possible entries and combining them into new tests.
  (b) 'CapBak' [252] captures keystroke sequences for automatic playback. CapBak includes screen-save capabilities, replay timing adjustments, and facilities to edit captured key-save files. Dynamic playback programming is provided by the use of IF and WHILE clauses in the keysave files.
  (c) CARBONCopy [252] is a terminal I/O capture program. Terminal keystrokes are recorded to a file where they can be replayed, edited or printed.
  (d) 'Check*Mate' [252] can perform individual tests of new functions using

keystrokes capture or manual coding depending on the complexity of the test. By using keystroke capture, testing operations need to be performed once; then they can be replayed to test the function again.

(e) 'Evaluator' [252] has a record mode where scripts are automatically recorded. In replay or playback mode, Evaluator replays the recorded keystrokes from the recording session. Playback mode can run unattended and save the results to files. In programming mode scrips may be edited in the TEST Control Language (TCL).

(f) TRAPS [252] is menu-driven and allows recording, editing and replay of test scripts.

- Capture-replay-comparator regression testing tools.

(a) 'Autotester' [252] is capable of testing applications on PCs, minis, and mainframe computers. The link is through asynchronous communication. It records tests and allows easy editing and playback capabilities. Autotester supports a structure method that promotes test modularisation and documentation. Procedures (scrips) may be used over and over again to test similar functions that occur at different times during a test session.

(b) DCATS [252] provides a method of writing a test script and inputting the expected results in order to record scripts. This script can then be executed and the results compared to the expected results. Difference in actual and expected outcomes are reported.

2. **Test Coverage Monitor**

This group contains tools that monitor test case coverage and keep track of which parts of a program are executed when a given set of test data is run. This involves executing an 'instrumented' version of the program with the test data provided. Test monitors can enhance a programmer's understanding of a program by identifying the code segment associated with particular user oriented functions.

They address the following important aspects of program testing :

- preparation of test data.
- measurement of test data coverage.

They should :

- compile and run programs with all available test data.

- identify and report any logical decision path or executable statements within the program that the test data fails to exercise.

With the information from a test coverage monitor, programmers can evaluate areas of a program that requires further testing, and software quality assurance personnel can make more accurate and reliable judgements of a program's readiness for production implementation.

(a) IITS [208] is a test coverage monitor and a regression testing tools. It is an utility system that can be integrated with the output of a programming environment to test and validate the results of the development efforts. The test coverage analysis tool guides development of test suites, assess testing progress, and aid error detection. The regression testing tool tests and retests candidate systems' functionality.

(b) ISAS reports and charts procedure hierarchy, data references, control flow and system structures.

(c) SMARTS [180] with EXDIFF execute, evaluate and report on thousands of tests automatically. Both interactive and batch tests are scripted in easy to maintain test files. SMARTS and EXDIFF, plus CAPBAK (see regression testing tools), plus if necessary, a 3270 emulation back end, forms a powerful system of tools for planning, executing, logging, and analysing complex repetitive test suites.

(d) TCAT [251] measures test thoroughness in terms of logical branchs, instead of statement coverage that common profilers use.

4.2.4 Tools for Maintenance Management

Two categories are identified to be useful for the management of maintenance: software configuration management and product management.

1. **Software Configuration Management**

   The purpose of configuration management [250] is to ensure that, at all times, the status of all versions of all development products is known and that the location of all copies is known; it is particularly required that the status of all shared objects is carefully controlled and that unauthorised changes are prevented. In addition, customers standards may require that other records are maintained, for example, detailed explanations of the reasons for changes in passing from one version to the next.

The controlled intermediate products (baselines), and the milestones at which they are established, form a vital unifying link between the control of the software development process and the control of the software product. The status of, and the access to, the intermediate products are strictly controlled but configuration management also means that:

- only the controlled versions can be used as input to other activities
- any proposed changes to the baselines must be processed through the formal change control procedures.

Existing configuration management tools range from simple tools such as Unix version control to complete configuration management systems such as CCC, Lifespan, and to integrated programming environments with software configuration management support built in like DSEE [138, 139], Adele [?], Aspect and ISTAR.

Configuration management can be seen to have three main functions [250]:

- Version and Variant Control which identifies all relevant items and records the history of their development through successive versions, permitting this history to be used retrospectively and previous states of parts of the project to be restored selectively. Version control is responsible for ensuring that all items are reliably stored, so that this restoration is always possible. Effective version control is necessary to support all other configuration functions. SCCS [211], CMS and RCS [261] are typical of simple change and version control tools with the function to manage and control the changes made to individual components.
- Configuration control which is concerned with the building of appropriately structured systems from their constituent parts.
- Change control which is the operation of applying changes, with suitable authorisation to establish new states though which the project passes. The authorisation required to establish a change allows the continuous application of quality assurance procedures throughout the project.

For more details of Software Configuration Management tools, see Appendix A and B.

2. **Product Management**

   As mention in Appendix A, three tools are commercially available for managing maintenance products:

   (a) SABLIME [115] is a comprehensive product administration system that tracks changes to a product consisting of software, hardware, firmware, and/or documents, from its origination, through maintenance, delivery, and support. Its integrated Modification Request (MR) and Configuration Management capabilities make it a unique tool for managers and product developers alike.

   (b) RA-METRICS [115] and SMR [115] provide a software metric repository. RA-METRICS supports all of the management reporting metrics and it reports: functional and technical quality, user satisfaction, defects counts, CASE/Tool Usage, development and maintenance history, financial history and estimation accuracy whereas the Software Metric Repository (SMR) is a menu and mouse driven database featuring a 'point and Shot' user friendly interface. The database incorporates the software metrics generated by PC-Metric as well as Functions Points and project data. The browse and reporting capabilities are encouraging to examine and analyse the raw data. PC-Metric is a software metric generation package. It analyses the source code and computes numerous size and complexity metrics.

4.3 Prototypes and Research Projects

Prototypes and research projects for maintenance are divided into four categories: program comprehension, fault localisation, impact analysis and knowledge-based systems.

4.3.1 Prototypes for Program Comprehension

In this section on prototypes for program comprehension, we give some examples that use code analysis and other techniques in order to comprehend the code.

(a) ASAP (Ada Static Source Code Analyser Program) is an automated tool for static code analysis of program written in the ADA programming language.

The purpose of this analysis is to collect and store information e.g. compilation unit's size, complexity, usage of ADA language constructs and features, and static interface with other ADA compilation units.

Another example is the prototype ISMM for incremental static analysis of C programs.

(b) The goal of ISMM is to demonstrate the feasibility and practicability of using incremental static analysis to aid in the maintenance phase of the software life cycle. ISMM [221, 222] consists of two modules: FREND, a front end which parses the C source code and convert it into an annotated directed graph representation of system calling structure, and BEND, a back end which performs both the incremental and exhaustive analysis.

(c) A prototype from IBM [49] combines a data base to store the program with a display 'viewer' that allows a programmer easily to browse through it in many ways to accumulate information for a maintenance task.

(d) MICROSCOPE [4] is an ambitious program analysis system using static and dynamic analysis of Common Lisp and Common Objects. This prototype permits different view of the source code and perform impact analysis, and also records a browsing execution history in a knowledge base.

(e) PUNS [50] is a Program Understanding Support Environment that gives multiple views of the program and a strategy for moving between views and exploring views in depth. It comprises two components, a repository and a user interface.

(f) AEGIS is used to maintain very large Navy weapons control system using the method to capture a large volume of data about the components of the software in a data base that can be queried or from which reports can be printed.

(g) SCORE/RM [52] provides a mechanism by which a maintainer can systematically work through the code and comprehend its purpose, produces a set of documentation to reduce future learning curves and modify the code so that it becomes easier to maintain.

(h) The University of Linz [224] has produce a prototype to helps programmers understand object-oriented software systems written in C++. It enables its users to easily browse through the system based on the relations among its classes, files and even identifiers.

4.3.2 Prototypes for Fault Localisation

Error localisation [131] in program debugging is the process of identifying program statements which cause incorrect behaviour. According to some researchers, the error locating process represents 95 percent of the debugging [174]. Examples of debuggers have been given in section 5. Currently, many techniques and tools are used to perform fault localisation and these methods can be classified as either knowledge-based or non-knowledge-based. Knowledge-based fault localisation systems can be identified by their autonomous behaviour. The system themselves interpret the information they generate to localise faults; the information is not passed to a user for interpretation, as is the case in a non-knowledge-based system.

(a) PROUST [118] is a knowledge-based fault localisation system designed to create a framework sufficient to catch all possible errors in small programs. The aim of PROUST is to understand the nature of the errors, state the errors, and suggest a form of solution. To accomplish these objectives, the system requires that the program be totally and correctly specified. The major limitations of this system is that it is extremely difficult to form such specifications even for small programs, and there is no way to guarantee the specifications are correct or complete even after they have been stated.

(b) PTA [42] is a Knowledge-Based Program Testing Assistant developed by Chapman. As programs are developed and tested, a user can request that the system automatically store the test cases for future use. When an error arises in feature being tested, the system in coordination with the user can request that the appropriate saved test cases be rerun automatically -either before the system has been repaired to aid in identifying the problem or after the system has been repaired to ensure its correctness. In conjunction with this capability, the PTA heuristically modifies the corresponding test cases when the source code is changed. This preserves the ability of the system to continue to use, if possible, previous test cases to perform a type of automated regression testing of the code .

(c) PELAS [131] is an error localisation assistant system which guides a programmer during debugging of Pascal programs. The system is interactive: it queries the programmer for the correctness of the program behaviour and uses answers to focus the programmer's attention on an erroneous part of the program (it can localise a faulty statement). This system uses the knowledge of program structure represented by the dependence network used by the error locating reasoning mechanism to guide the construction, evaluation and modification of hypothesis of possible causes of the errors.

New techniques for fault localisation are currently being developed:

(a) Collofello and Cousin [63] developed a theory called relational path analysis that suggest that there exists information associated with the execution paths of programs which when analysed heuristically can produce statistically significant fault localisation.

(b) Korel and Laski [132] presented a novel fault localisation algorithm that is capable of identifying a restricted class of programming faults for the Pascal language. The algorithm uses the computation trajectory-based influence relations to formulate hypotheses about the nature of the fault and user input is needed to asses correctness of intermediate situation on the trajectory. As the complexity and size of software systems continues to increase dramatically, emphasis is needed on developing automated methods [54] to help perform fault localisation an repair activities.

(c) Shahmehri [231] presented a semi-automatic error localisation method for Pascal, Fortran or C with side-effects. They are using program slicing and a data flow analysis technique to dynamically compute which parts of the program are relevant for the search. A prototype error localisation has been implemented in Pascal.

4.3.3 Prototypes for Impact Analysis

Impact analysis evaluates the effects of a proposed change. This activity determines whether the change can be made without adversely affecting the rest of the software.

Traceability suggests the connectivity of the relevant Software Configuration Items

(SCI) and whether traceability can be established once the proposed change is made. The design of modification requires an examination of the side-effects of changes.

1. **Traceability**

   Some attempts have been reported at examining a change before its implementation. Some examples are given: Requirements traceability has been extended beyond the traditional tracking of requirements to making predictions of the effects of changed requirements [201]. Honeywell's Requirements to Test Tracking Systems (RTTS) tracks the Navy's documents specified by MIL-STD-1679. RTTS creates, analyses, and maintains traceability links among life-cycle documents. Other systems using traceability are: SOFTLIB, GENESIS, CMVT, NSE.

   The maintenance process is viewed by Pfleeger [197] in terms of the software workproduct as a graph of software life-cycle objects connected by horizontal and vertical traceability. The former (dependencies analysis) addresses the relationships among the parts of the workproduct. The later addresses the relationships of these components across pairs of workproduct. Both types of traceability are necessary to understand the complete set of relationships to be assumed during impact analysis.

   Vertical traceability has been addressed, but is still restricted to the source code.

   (a) The University of West Florida [273] is building a dependency analysis tool set with a dependency analyser and a tool for building comprehension tools. The intent behind this tool is to provide a basis for determining program dependencies (data flow, calling, functional and definitional), so by creating the application specific front-end, the comprehension aid can be tailored.

   (b) The University of Naples [47] is using reverse engineering and static code analysis that enable the identification of the actual and potential intermodular dataflow relationship.

   Some software development environment have incorporate horizontal traceability as part of their overall approach to development [197] e.g. ALICIA, SODOS [105, 106], PMBS, and DIF [176, 86].

   (a) ALICIA addresses Software Life-cycle Objects (SLO) granularity using information content rather than the entire documents. It support completeness and consistency checking of traceability relationships as well as navigation among

SLOs in the project database.

(b) SODOS supports the development and maintenance of software documentation. It manages SLO using an object-oriented model and a hypermedia graph of relationships. The documents are defined in a declarative fashion using a structural hierarchy, information content, and intra/inter document relationships and navigation among SLOs.

(c) DIF is a hypertext-based documentation integration facility that provides a mechanism for developing and maintaining software documentation with its associated relationships. It enables visualisation of objects in the software system, hierarchy charts of software objects, and display the dependencies of related software objects.

2. **Ripple-effects**

Ripple effect is the phenomena by which changes to one program area have tendencies to be felt in other program areas. Some examples are given:

(a) Arizona State University [54] have done research on a ripple effect analyser and have implemented one in prototype environment.

(b) Surgeon's Assistant is a prototype from Gallagher that slices up programs, extract pertinent information, and displays data links and related characteristics so the user can track the changes and influence on targeted structures. It delivers semantic information and editing guidance to help the user to formulate a maintenance solution with no undetected link to unmodified code, thereby eliminating the need for regression testing.

(c) The Software Engineering Research Center [281] is producing a prototype system for a reduced set of Ada languages that has been implemented to demonstrate the usefulness of logical ripple effect analysis.

4.3.4 Prototypes for Knowledge-based Systems in Maintenance

A Knowledge Based System contains large amounts of expert knowledge which can be brought to bear on a given task. An expert system is a species of knowledge based systems which has built into it the knowledge and capability that will allow it to operate at the expert practitioner's level.

A knowledge based system essentially consists of:

- a knowledge base containing facts, rules, heuristics, and procedural knowledge, acquired and stored declaratively in a basical random order.
- an inference engine, which consists of reasoning, problem solving and research strategies.
- a user interface for the dialog with the user.
- an explanation generator, which is a set of procedures dedicated to answering such questions as why a goal was met, or how a set of assumptions might lead to alternative conclusions.

There has been little reported work on the use of knowledge based systems in software maintenance:

(a) Cross [64] described an Expert System approach to building an information/maintenance tool for an existing target system of both hardware and software components. The purpose of tool is to help the user identify the components they seek and to automate the identification of the remaining supporting components required. The tool uses its rules rules-based knowledge and the user selections to identify the desired components and their supporting components.

(b) Pau and Negret [190] described a software maintenance knowledge based system called SOFTM which was designed for the following purposes:

- to assist software programmers in the application code maintenance task.
- to generate and update automatically software correction documentation.
- to help the end user register, and possibly interpret, errors in successive application code versions.

SOFTM relies on a unique ATN (Augmented Transition Network) based code description, a diagnostic inference procedure based on pattern classification, and on a maintenance log report generator. The system is able to a range of programming languages provided that code descriptors can be extracted from the code. SOFTM has 3 types of knowledge base:

- Facts about error types, error locations, diagnostic classes, and the environment.
- Code independent rules that apply to the general software maintenance task.

- Symbolic descriptors derived by rewriting, in predicate form, features of programming languages provided by the compiler, the specification language, or the data flow model.

(c) Calliss, Kalil, Munro and Ward [40] described an intelligent, knowledge approach to software maintenance by describing a tool that is intended to help reduce the amount of time spent analysing code. They have identified 3 types of knowledge:

- Maintenance Knowledge: this is the knowledge about how the maintenance programmers do their work and is elicited from expert maintainers. This knowledge provides the bulk of a systems heuristic knowledge that dedicate the weighting patterns on searches through the expert system.
- Program Plans: they are two different categories of program plans:
  - General program plans with a small set of plans that show commonly occurring activities in computer programs.
  - Program class knowledge with a set of plans common to a particular type of program.
- Program Specific Knowledge: this is the internal representation of the source code together with knowledge obtained from using static code analysis tools such as cross referencers, data flow analysers, call graph generator, etc.

4.4 Summary

Currently, there is research on software maintenance tools but which is far smaller than for development tools.

At this time, no one method with tools and environments has succeeded in integrating the diversity of maintenance tasks, tools and situations in a consistent way. Tool support is usually restricted to a single phase of the maintenance process. There is a lack of maintenance tools like the concept of intelligent maintainer's assistant that could be semi-automatic and could help the maintainer during the whole software maintenance process from identification of modification to revalidation.

Chapter 5

Integrated Project Support Environments

5.1 Introduction

The objectives of this chapter are to define an IPSE with its features, to describe the requirements for the next generation of software in the aerospace industry, and to evaluate current IPSEs according to some of these requirements.

5.2 What is an I.P.S.E. ?

5.2.1 Introduction

The IPSE arose from the observation that, in many projects, useful data was generated and used by the project tools set as part of the development process. However, it was difficult to get tools to exchange data because they used incompatible data formats. These tools gave only partial, fragmented support to the managerial, technical, and administrative tasks involved in the development and maintenance process.

It was recognised [161] that a profusion of tools alone is not enough to instill good engineering practices in the industry; what is required is a stable, consistent and integrated approach to the working environment. Thus, the Integrated Product Support Environment (IPSE) concept was born.

An Integrated Project Support Environment [259] is an integrated environment that focuses on the developmental aspects of the software life cycle. It may not include all phases, e.g. maintenance.

5.2.2 Features of an IPSE

This section describes the features and structure of an IPSE. At the logical centre of an IPSE there is a basic set of facilities termed a kernel. An extra set of facilities may be provided which extend those of the kernel and built using kernel facilities. The user can use both the facilities of the kernel and the extended facilities and these together constitute the infrastructure.

The tools of an IPSE will be implemented using the facilities of the infrastructure. The kernel, together with the infrastructure and tools constitute a populated IPSE. In some IPSEs, it may be possible to extend the kernel by incorporating user tools into it.

IPSE products provide facilities in four main areas [250]: Human Computer Interface, Data Management, Activity Management and Integration of tools

- The Human Computer Interface (HCI) or Man Machine Interface (MMI), or User Interface (UI) is probably the most important factor in producing an IPSE that will be acceptable and therefore used. The concept of an IPSE is that it will be used by all project staff whether engineer or manager. Therefore the users should have a clear conceptual model of the IPSE and the information it contains and that the style of the tools should be dictated by this model and by the requirements of the tasks at hand. So, the IPSE needs to handle the user interface service, instead of leaving it to each tool. HCI facilities should be varied, flexible and multi-windowed with menu, mouse and command line. The kind of interface required may be different for the different types of user and different types of activities.
- The IPSE Data Base needs to contain the repository of all project information: requirements, designs, code, test cases, documents and other relevant information. The data base should support modelling of all aspects of data associated with the development process and should be extensible and configurable to allow extension and adaptation of the IPSE as a whole to individual project requirements. The infrastructure of the IPSE should support software configuration management.

- The Activity Management is the ability to model and manage specific activities associated with project procedure and methods. It is composed of the means to define a process structure, the facilities to define the order and type of tool invocation, and the facilities to model methods specific processes.

  For example, after a file has been edited, the configuration control tool should be invoked.
- The integration of tools can be seen through a Public Tools Interface (PTI) and a Foreign Tools Interface (FTI):
  - The PTI is an interface for tools written specially for the IPSE.
  - The FTI is an interface for existing or commercial tools that we want to integrate in the IPSE.

5.2.3 The Ideal IPSE

The ideal IPSE [250] according to Aerospace criteria will provide a context for software development over the whole life-cycle and will:

- be capable of supporting the development of large, complex and high quality software
- provide controls and facilities to allow projects to be carried out with optimum productivity and quality
- be fully integrated so that information can be freely exchanged within the project
- provide all users with an appropriate and tailored set of facilities with which to carry out their tasks

Across a company and its project, a variety of different tools, methods and procedures are used, some tailored to specific project needs. In addition, the need of projects may well change and evolve as new methods and tools are developed and are made commercially available.

Version control and configuration management are critical to ensuring the consistency and reproduceability of delivered systems. It is important that support is given, in the IPSE, for these fundamental activities.

Change control must be supported by the recording of dependencies and tracing information within the IPSE database. This must be done across the whole life-cycle by providing the relationship between design and requirements, code and designs, document and code.

Thus, the ideal IPSE will need to be extensible and adaptable to individual needs.

5.2.4 Conclusion

The ideal IPSE is a blend of:

- flexibility, giving users the capability of tailoring the IPSE to their requirements and ensure that its capability will accommodate any changes in these requirements as the project evolves. It will ensure that all the activities in the project can be supported.
- integration, ensuring that the users have a consistent and uniform view of the IPSE.

However, no IPSE so far has approached the ideal blend with flexibility and integration

5.3 Requirements for Software in the Aerospace Industry

5.3.1 Introduction

The next generation of software in the Aerospace industry will be large and complex, developed by many groups of people at many locations, and will be expected to operate safely and undergo extensive evolution over its lifetime.

Requirements for software maintenance in the Aerospace domain [238] will increase for the following reasons:

5.3.2 Safety Critical Systems

A system or sub-system may be described as safety-critical if there are potential consequences of using the system that are so serious that it cannot be used at all unless the probability of a high-cost event (an accident) occurring is very low. For example, a system is usually considered safety-critical if some behaviour of the system can result in death, injury, loss of equipment or property, or environmental harm. When computers are used to control safety critical processes, there is need to verify that the software will not cause or contribute to an accident [41].

Some aerospace systems will become safety critical with manned space flight like the launcher ARIANE 5, the space plane HERMES and the station COLUMBUS.

Therefore, the IPSEs that will support the development of the software should provide specific support for the development and verification of safety critical software, and for fault tolerant software development.

5.3.3 Increasing Software Size

Software size will increase by a factor of ten. Currently less than a million lines of code is used for a satellite ground system while few millions lines of code for HERMES, COLUMBUS.

The IPSE should support the development of very large scale software.

5.3.4 Increasing System Lifespan

Software lifespan will increase from less than 10 years to more than 20 years. System architecture will need to be flexible in order to take into account new technologies.

Therefore, the extensibility of the IPSE is a major criteria as is the integration of the new tools in the environment.

5.3.5 Distributed Developments

The trend toward the development of large integrated software systems in short time scale leads to the need for a distributed development capability. Many large projects involve collaborative working between partners in a consortium, and suppliers and subcontractors, often on a European or International basis.

Software maintenance methods should take into account the fact that aerospace data processing projects will involve contributions from several subcontractors.

HERMES is dominated by France, COLUMBUS by Germany, and the remainder of the 13 ESA countries contribute with very different scales to each project whereas some countries contribute to only one of the two projects.

Because ESA (European Space Agency) follows the fair return principle, that means a participating country receives industrial work in relation to its contribution, and as projects have the responsibility to follow this principle it is extremely difficult to release financial responsibility from a project for a common development. In this case, how can maintenance be adequately performed ?

In situations where the purchaser requires that the software be maintained by the purchasing authority, or a third party, then the overall development must be controlled to produce a unified product that appears to have been generated by a single team.

This mean that all contributors must [250]:

- use identical tools and environments
- produce consistent documentation
- operate standards procedures
- use standard reporting formats
- deliver all design documentation and tools data

The software will be developed in several geographical locations using the same IPSE facilities and communication between the IPSEs is required. Wide areas communication based on commercial products should be provided to support remote logging and file transfer from one site to another. The concept of remote maintenance for corrective activities should be investigated and an important attribute of distribution is *granularity* which is the size of object that can be exchanged between machines or operational sites.

5.3.6 System Perenniality

With the increasing system lifespan, system perenniality is becoming a major problem for both hardware and software.

Software environments are using tools that are primarily commercial off-the-shelf product and the perenniality of these tools must be assured before buying commercial tools. The stability and maturity of both the commercial tools and its supplier must be clearly established to keep the product operational during its lifespan. The product manager must ensure that projected functional releases of the tools accord with the release of the product.

Therefore, the concept of extensibility of the IPSE to incorporate extra facilities should be evaluated as well as the integration of these tools into the IPSE.

5.3.7 Reuse

The development of aerospace products requires intensive development of large complex software system and for these reasons, special attention [75] is currently devoted to the minimisation of effort and thus to cost reduction.

Therefore, the IPSE should provide an environment to support reuse of software elements at different level of abstraction (e.g. specification, design, code, documentation ...). This is a field of much current research.

5.3.8 Training and Knowledge Transfer

The loss of an employee is very damaging to a company's prospects. Adequate training and knowledge transfer will continue to be required as there is no evidence that the software engineering staff turnover rate will slow down in the near future.

The IPSE should provide training facilities for each tool and for the environment.

5.3.9 Conclusion

It is not possible to evaluate all current IPSEs and this is not the objective of this thesis. Therefore from the requirements, some criteria remain the same like integration, flexibility and distribution (see Table No 5.1).

Requirements for Aerospace Systems	IPSE should support	Criteria for evaluating current IPSE
Safety Critical Systems	Safety Critical Systems	
Increasing Software Size	Very Large Scale Software	
Increasing Lifespan	SCM	Flexibility
Distributed Development	SCM	Distribution, Communication
System Perenniality		Extensibility, Integration
Reuse	Reuse	
Training	Training	

Table No 5.1: Prospective requirements and their effects on the IPSE

5.4 Criteria for Analysing IPSEs

5.4.1 Introduction

As mention in the previous section on requirements for the aerospace industry, we need to define the three criteria for analysing current IPSEs: flexibility, integration and distribution.

5.4.2 Flexibility

Flexibility is the measure of how easy it is for the user to adapt an IPSE to support the activities of a given project. This can be done in two ways

- by extending the facilities of the IPSE
- by tailoring the existing facilities of the IPSEs to particular project needs (instantiating a generic IPSE).

1. **Extensibility**

   Extensibility is the ability to incorporate extra facilities into the IPSE. Two categories of tools can be added: native and foreign tools. The former are built especially to work in the IPSE and fully integrated through a Public Tools Interface. The latter might be rehosted from a foreign environment and therefore, the interface will need to be adapted to those of the IPSE.

The IPSE might allow the kernel to be extended, for example by enabling extra facilities to be incorporated or defining a new type in the data base.

2. **Tailorability**

   Tailorability is the ability to adapt the existing facilities of the IPSE so as to provide support for a project.

   A project involves a range of different structures and activities.

   - The structures are those associated with the product and those associated with management.
   - The activities are those associated with project procedures such as coding, configuration control and project management.

   An IPSE is supplied with the basic set of facilities from which support for a project need to be fashioned according to the wide range of users from programmers to managers.

   Tailorability is an assessment of the extent to which support for project activities can be constructed, the range of structures that can be reflected into the IPSE and scope for building interfaces suited to particular user needs.

5.4.3 Integration

Integration in an IPSE is the degree to which such things as common definitions and uniform styles of interaction are supported and facilitated.

There are three facts of integration:

- user interface
- data management
- activity support

1. **User Interface**

   Integration of the user interface is related to the way a user interacts with the IPSE. A user requires a consistent style in the user interface of all tools and to the IPSE infrastructure. The style of presentation and responses to such things as menus, icons and function buttons must be uniform.

2. **Data Management**

   Integration of data management does not simply imply that tools share a common data base. It is that the structure of the data is also held separately, rather than being implicit in the tools themselves. The IPSE data base should provide a uniform means of accessing and manipulating the data for both direct user access and tool access. The data base may also include facilities to enforce any consistency constraints specified for a project. The smallest size of object that can be referred to in an IPSE will have a bearing on the degree of integration.

   Thus an IPSE which only allows reference at file level will still leave the interpretation of the format of the contents within the file to individual tools.

3. **Activity Support**

   Integration of the management is the way in which tools are interrelated, the way the development process can be moulded into a logical whole, rather than comprising a series of independent tasks.

5.4.4 Distribution

Distribution covers both:

- The physical aspects of the system (Hardware and Software)
- The logical architecture of the IPSE itself

1. **Physical Aspects**

   The physical aspects are concerned with whether the facilities of a single IPSE are accessible over Local Area Networks (LAN) or Wide Area Networks (WAN).

2. **Logical Aspects**

   The logical aspects are concerned with the way in which the underlying database and kernel facilities are shared across different machines.

5.4.5 Conclusion

IPSEs should be able to incorporate new facilities and to adapt the existing one to projects during the software life-cycle they support. The integration level of tools and user interface is of critical importance to obtain a consistent IPSE. The distributed development requires information shared across different machines and accessible over networks.

The above criteria are used to evaluate the current IPSEs.

5.5 Evaluation of IPSEs

5.5.1 Introduction

IPSEs have only been available for a short time. The DTI and NCC have made an evaluation of IPSEs [250]. They assess different IPSEs e.g. ALS, ASPECT, BIS-IPSE, DSEE, Eclipse, EPOS, Genos, ISTAR, Maestro, PACT, PCTE Emeraude, Perspective, Perspective Kernel, Prados and Rational. Some of these IPSEs will not be evaluated because they do not conform to the criteria on prospective requirements:

- o ALS, ASPECT, BIS-IPSE, DSEE, EPOS, Maestro, Perspective and Prados are limited by the size of the project.
- o BIS-IPSE is primarily designed to support the design and development of commercial Data Processing systems.
- o Perspective is language oriented (Pascal).
- o Rational provides a program development and support environment for users of the Rational range of computers.

The IPSE that will be evaluated are the following:

(a) Eclipse

Eclipse is PCTE based initially supporting MASCOT 3 (a method for designing and building software aimed at real-time embedded systems) and LSDM (a method for structured system for analysis and design). It is being produced to run under the Emeraude implementation of PCTE.

Eclipse is a British project which chose the PCTE interfaces on which to develop a set of general mechanisms available to tool writers; a tool set built using these facilities; general tool builder support facilities, and a research program. Its objectives are centred around the use of an advanced database system specifically written to address the data management issues associated with software development.

(b) Genos

Genos provides a distributed open and incremental integrated environment for the development of software projects.

(c) ISTAR

ISTAR is an Integrated Project Support Environment.

(d) PACT

PACT is an ESPRIT project that is building an integrated environment on the PCTE interfaces. It aims to provides an integrated toolset.

(e) PCTE Emeraude

EMERAUDE is a project undertaken by a French consortium of three companies, to produce an industrial quality implementation of the PCTE interfaces.

(f) Perspective Kernel

Perspective Kernel provides a general IPSE infrastructure incorporating the ideas and tools of Perspective, but open to new tools and methods.

5.5.2 Evaluation of IPSEs with the criteria

1. **Flexibility**

   - **Extensibility**

     (a) PCTE Emeraude is very flexible allowing new tools to be incorporated. The purpose of PCTE as a tools interface and the movement of it toward standardisation is to encourage the third party production of compatible and integrated tools.

     (b) Eclipse and PACT are open IPSEs based on PCTE. PACT with the Common Service layer provides a PTI above the level of that provided by PCTE. The facilities provided by the the PACT kernel and PCTE are extensible by the definition of new schemas and data structures. Eclipse can integrate new native tools with PCTE and is fully compatible with most Unix tools.

(c) Genos has the capability of integrating new tools very easily either through the external tools interface or by 'encapsulation' depending on their interaction with the operating system.

(d) ISTAR native and foreign tools can be integrated using the ISTAR tool building tools. To ensure that foreign tools can exploit the database and will reflect ISTAR's user interface conventions, the foreign tools are packaged in an 'envelope'.

(e) Perspective Kernel has the capability to integrate native and foreign tools. With the former, it will be through a native tool interface whereby tools can be written to take advantage of the facilities of the kernel directly. With the latter only with VAX/VMS tools written without knowledge of the kernel. The kernel data structures will themselves be extensible.

- **Tailorability**

(a) Eclipse supports the contractual model of project control in its configuration control system. Data base structure is described by schemas that can be modified and added. There is a facility for defining the appearance of the end-user interface based on an interpreted language. The user interface and the kernel structures are configurable by the tool builder but not by the end-user.

(b) Genos has a flexible way of modelling project process so that tool invocations are integrated with project processes.

(c) ISTAR can be tailored to project needs by structuring contracts and supplying suitable workbenches. The product structure can be reflected into the structure of contracts. The user interface can be tailored and forms and menus generated for specific applications.

(d) PACT Common Services provide extended support for the composition of tools. Generic tools can be produced to operate on data objects in the PCTE Object Management System (OMS).

(e) Extensive support is provided by the PCTE OMS and the tailored features of the configuration management tool for the definition of product structure, although no explicit structures are defined be PACT.

(f) PCTE Emeraude is capable of capturing a very wide range of data structures and no particular development process is assumed. Data is described by easily modifiable schema. It has no explicit support to enable an end-user configurable interface.

(g) Perspective Kernel will be tailorable by tool composition, extending the data base structure and by defining an appropriate project structure. The data base will have a type structure with inheritance allowing generic tools to be provided and specialised for particular organisations, projects and applications.

2. **Integration**

- **Integration of the user interface**

(a) Eclipse has a high level interface and therefore native Eclipse tools are highly integrated as regarded user integration.

(b) Genos has a uniform interface used throughout by all tools and Perspective kernel will provide a uniform one for native tools.

(c) PCTE has graphics capabilities and pop-up menus that present a consistent user interface.

(d) ISTAR provides a uniform set of terminal independent logical services for all styles of interaction.

(e) PACT will provide and enforce a uniform and coherent user interface through the use of PCTE facilities and PACT Common Services to implement all integration between tools and the user.

- **Integration of data management**

(a) Eclipse native tools store all their data in a highly structured way within the data base. The database interface is at a high level and supports retrieval by pattern matching and simple value comparison. In this way information retrieval by different tools is highly integrated and there will be no inconsistencies when logically equivalent data is shown to the user.

(b) Genos supports typed data and allows tools to access common data structures, so that generic tools can be produced to operate on common types of data from within the user interface. For the data base implementation such as PCTE a more complex data model is supported which allows tools access to relationships between data objects and more extensive attribute information.

(c) With ISTAR, data types can be defined and shared via workbenches. PACT integrates data in a consistent way with the use of the PCTE OMS and the PACT Integration Rules.

(d) With PCTE Emeraude, foreign Unix tools are not really integrated in their data management and native tools have not yet been built. However there are mechanisms which allow the integration of foreign tools to be improved.

(e) Perspective kernel will integrate both native and foreign tools via shared data.

o **Integration of the activity management**

(a) Process management is very closely integrated into Genos in the support for project structures, so that the use of particular tools can be integrated into the development process.

(b) With ISTAR, standard procedures can be defined as scripts and shared via workbenches.

(c) PACT provides some support for tool composition through the Unix shell component of the initial tool set and some further support for tool composition using the OMS is planned.

(d) PCTE is similar to Unix, but can also represent the relationship between an interpreter and its code on the database.

(e) Perspective kernel will provide some degree of process integration via the process structure and transaction facilities.

3. **Distribution**

o **Physical Aspects**

(a) PCTE is designed as a distributed system supporting a network of workstations distributed over an Ethernet LAN of heterogeneous machines. The granularity of distribution is an object.

(b) Eclipse and PACT are potentially distributed because they are PCTE based.

(c) Genos exploits both LAN & WAN distribution over heterogeneous systems. For existing systems the granularity of distribution is at the level of a file so that Genos can take advantage of distributed file systems. For data base implementation the granularity of distribution is an object.

(d) ISTAR can make use of both LAN & WAN because of its contract structure. However, a contract is usually restricted to one host while subcontracts can be on other hosts. The machine within the network need to be of the same type as long as they exchange data over the network.

(e) Perspective Kernel is not a distributed system but is available on VAX clusters.

- **Logical Aspects**

(a) PCTE distinguishes process distribution and the data base distribution of objects. Facilities exists that enable the user to control the distribution.

(b) With PCTE, PACT and Eclipse, the distribution of objects is invisible to the end-user.

(c) The logical model for distribution by Genos is based on the project view from which projects are built. Within a single project, the project views may be distributed over a number of systems.

(d) With ISTAR, contracts can be allocated to particular hosts with the data associated with the contracts. A particular host may have a particular set of workbenches to carry out its contracts.

(e) Perspective Kernel is not a distributed system but is available on VAX clusters.

From the evaluation of current IPSEs according to the above criteria, a summary is given in Table No 5.2.

EVALUATION CRITERIA	Integrated Project Support Environments					
	ECLIPSE	GENOS	ISTAR	PACT	PCTE E.	PERSP.K.
FLEXIBILITY						
1. Extensibility						
for Native tools	PCTE			PCTE	PCTE	PTI
for Foreign tools	Unix tools	FTI or 'enc.'	'envelope'	PCTE	PCTE	Vax/Vms
for Data Base				data struc.		data struc.
2. Tailorability	project control	project process	structuring contracts	tool composition		project structure
	data base		workbench	SCM	data base	data base
INTEGRATION						
1. User I/F	high level	uniform	uniform	uniform & coherent	consistent	uniform
2. Data Mgt.	support retrieval	data type	data type	data type		share data
	data struc.	data struc.				
3. Activity Mgt.	?	Yes	standard procedure	tool composition	like Unix	process structure
DISTRIBUTION						
1. Physical Asp.	LAN	LAN&WAN	LAN&WAN	LAN	No	
2. Logical Asp.	invisible	Yes	contracts	invisible	invisible	No

Table No 5.2: Summary of the current IPSEs' Evaluation

5.5.3 Conclusion

As described in the previous section (evaluation of IPSEs according to three criteria) no IPSEs so far has approached the the ideal blend with flexibility and integration. Furthermore, ECLIPSE uses MASCOT and LSDM methods which are not supported by the European Space Agency's standards. GENOS uses no standard configuration management tools and does not supports a particular specification and design method. ISTAR is mainly dedicated to real-time systems with the use of CORE method for specification and SDL for design.

Perspective Kernel is not a distributed system. PCTE Emeraude is very interesting with the use of the European standard PCTE, but should use foreign tools that are compatible with PCTE. None of these IPSEs supports reuse of components.

For these reasons, the aerospace industry had to build their own IPSE to develop the software according to requirements of different projects.

5.6 IPSEs for Aerospace Systems

5.6.1 Introduction

In this section, an evaluation of IPSEs for aerospace applications is performed. The IPSEs used for the software development of HERMES/COLUMBUS and FREEDOM are integrating tools that are primarily commercial off-the-shelf.

5.6.2 HERMES/COLUMBUS

The Hermes Software Development Environment (HSDE) [27] and the Colombus Software Development Environment (Columbus is the European contribution to the International Space Station) both support methods and standards, allowing distributed development and management of large, complex and critical Ada software. The HSDE is dedicated to the development of the HERMES' software (European Space Shuttle).

The European Space Agency has provided appropriate standards, recommendation and infrastructure to ensure the successful and timely production of reliable software. These consideration has has led to the requirement specification of the European Space Software Development Environment (ESSDE) which for its life-cycle approach is based on the well established ESA Software Engineering Standards (ESA PSS-05-0, Jan 1987). The Columbus and Hermes projects [226] have used these requirements as input to their projects specific IPSE's and as a consequence many of the methods and tools are similar.

Due to the project autonomy, two separate teams (a Columbus team and a Hermes one) were established to produce the two IPSE's and although working to the same functional requirements to individual projects requirements were not sufficiently detailed to enforce commonality.

An evaluation has been made at ESTEC [226] of the first version of both IPSEs:

- all the commercial tools are different except for the requirement analysis and the syntax editor.
- all the developed tools are different in concept or implementation
- both environments present good and not so good concepts
- choosing one of the IPSEs's as the ESSDE would mean giving up the good concepts implemented by the other one

A combination of a the best concepts implemented in either IPSE will be used for the ESSDE.

5.6.3 FREEDOM

The Space Station Freedom Program has required the use of a common software engineering environment for the development of all its operational software, both flight and ground based. This environment, known as the Software Support Environment (SEE) [108], is really a large collection of tools, rules and procedures from several technologies. The SEE uses tools that primarily commercial-of-the-shelf, with limited capabilities being provided by custom tools.

The generic SSE [203] is an ordered collection of tools, rules and procedures which may be instantiated, or subsetted, to provide a wide range of software life cycle support systems.

5.6.4 Evaluation of Aerospace IPSEs

EVALUATION	NAME of Aerospace IPSEs	
CRITERIA	HERMES/COLUMBUS	FREEDOM
FLEXIBILITY		
1. Extensibility		
for Native tools	encapsulation	?
for Foreign tools	encapsulation	?
for Data Base	Yes	Yes
2. Tailorability	Yes	Yes
INTEGRATION		
1. User I/F	high level	high level
2. Data Mgt.	support retrieval	support retrieval
	object/sub-object	object
3. Activity Mgt.	Yes	Yes
DISTRIBUTION		
1. Physical Asp.	LAN&WAN	LAN&WAN
2. Logical Asp.	invisible	invisible

Table No 5.3: Summary of the Aerospace IPSEs' Evaluation

SEE and ESSDE are flexible and integrated environments, and a good distribution is provided.

SEE and ESSDE are built with tools that are primarily commercial-off-the-shelf. Therefore, it is difficult to maintain traceability across the software life-cycle between requirements, specification, design and test components. For example, outputs from specification are not compatible with inputs for design because the tools are designed to support different methods.

These environments do not provide tool support for program comprehension with error localisation and impact analysis, which is essential for maintenance, neither is there support for the costing of modifications to software.

These environments are not using standards for tools interface, thus it will be difficult to add foreign tools that will be fully integrated within the environment.

5.6.5 Conclusion

The 'construction' of an IPSE from existing tools [226], well established development concepts and a bit of glue software has proven to be surprisingly difficult. By building bottom-up, a universal multi-project IPSE could not be realised. There is a need to build these environments with a bottom-up and a top-down approach.

5.7 Summary

In this chapter, an evaluation of current IPSEs for development has been achieved and it seems that they do not support maintenance in an efficient way. The IPSE for the aerospace industry should support the development and maintenance of large and high quality software, running over many years in different locations. According to the requirements in the aerospace industry, there is a need to address the requirements for a software maintenance support environment.

Chapter 6

Requirements for a Software Maintenance Support Environment

6.1 Introduction

The previous chapters have discussed a method to perform software maintenance at three levels, described tools to support the software maintenance process and presented an evaluation of current IPSEs. As has been mentioned, these environments are addressed to development but do not fully support maintenance.

This chapter specifies the requirements for a Software Maintenance Support Environment (SMSE) for Aerospace software.

The SMSE should be customisable and extensible and should be able to support the maintenance of large to very large scale software. The SMSE should cover the full range of software maintenance activities described in the previous chapters. The SMSE should support a software maintenance process with sufficient precision and clarity to foster understanding, communication and effective support by means of tools. The toolset should include tools supplied by the development environment and tools dedicated for the maintenance phase.

The toolset should remain open-ended, extensible and should allow the addition of new tools through a standard Public Tool Interface. The SMSE should be flexible to accommodate change by permitting tool introduction, removal, replacement, customisation, extension, etc.

6.2 Data Base

The data base should act as a central repository for information associated with each project throughout the project life cycle. The data base should be extensible and configurable to allow extension and adaptation of the SMSE as a whole to individual project requirements. The data base should store information which allows management reports to be generated.

The Data Base Management System should store and retrieve all the data object produced by the project and should support software configuration management by retaining the currently approved versions of all controlled products.

6.3 Human Computer Interaction

The Human Computer Interaction (HCI) facilities should be varied, flexible and multiwindowed with menu, mouse and command line.

The HCI facilities should be tailorable to maintenance needs. The HCI should help and assist efficiently the maintenance staff during the operational phase. The kind of interface required may be different for the different types of user and different types of activities.

Two aspects should be considered:

- surface aspect: interface design must enforce ergonomical concepts (e.g. colour, screen object localisation, interaction mode, user background and task to be performed)

- architecture: object-oriented approach is widely used structuring HCI items

6.4 Software Configuration Management

The SMSE should support software configuration management with the following activities:

- software configuration identification
- software version control
- software change and configuration control
- software configuration status accounting
- software configuration audit

6.4.1 Software Configuration Identification

A software configuration item (SCI) is a 'manageable' software entity within a configuration e.g. requirement document, specification document, design document, source code, data file, documentation, test procedure.

Software configuration identification should support:

- definition of the different baselines and associated SCIs of a system
- identification of any change made to these components and baselines
- identification of the relationship between SCIs
- identification of the version of the tool that generated the SCI

6.4.2 Software Version Control

Software version control should:

- support version control for SCI

- track the historical record of each SCI
- store relevant information about each change performed on any SCI
- control multiple versions of the SCIs
- identify differences between two versions of a particular SCI including additions, deletions and moves
- correct the documentation and highlight changes made in a documentation

6.4.3 Software Change and Configuration Control

Software configuration control should:

- allow the partitioning of the software product in different geographical sites:
  - reference site
  - maintenance site
  - operational sites
- store required information to
  - create and control a new SCI
  - build and control a new release
  - modify a SCI
- manage a link between problem reports and SCIs
- manage a link between change request forms and SCIs
- manage change control forms on-line
  - use of standard format of the forms
  - creation of the software problem report
  - creation of the change request forms
  - progress of change
  - approval (signature)
  - report of modification

6.4.4 Software Configuration Status Accounting

Software configuration status accounting should provide an administrative history of the evolution of a software system by:

- reporting all configuration items including status of:
  - all software problems
  - all software documentation
  - all software release
  - all changes affecting the SCIs

  and generating information with statistics
- reporting the number of anomaly/modification request per category, site, SCI
- providing list of known errors and omissions
- identifying all SCIs potentially affected by a proposed change
- providing information of all software problems before a new release of a software product and the reasons for changes from one version to the next one

6.4.5 Software Configuration Audit

Software configuration audit should determines whether or not baselines meet their requirements and whether correct procedures have been adhered to.

6.5 Program Comprehension

The SMSE should support program comprehension with static, dynamic, impact analysis and traceability tools. These tools should be interfaced with the Software Configuration Management facilities.

6.5.1 Static analysis

The SMSE should support static analysis by providing:

- language audit:
  - conformance to project naming convention
  - absence of duplicate names
  - absence of non-standard language features
- control flow analysis:
  - absence of structurally unreachable code
  - absence of structurally non-terminating loops
  - absence of multiple entries to loops
  - conformance to recursion conventions
- data use analysis:
  - initialisation of data before use
  - use of all declared variables
  - absence of redundant writes
- information flow analysis:
  - the set of output variable
  - the set of input variable
  - the relationship between the above (which output variables may be affected by a change in a given input variable)
- symbolic execution (automated form of desk walkthrough in which execution of the code is replaced by symbolic operations)
- semantic analysis
- complexity measurement analysis:
  - code metrics e.g. program size, control graph, call graph
  - count of the independent logical paths through a procedure e.g. cyclomatic metrics
  - accessibility of a module, testability of a path, a system e.g. Mohanty's metric
  - amount of information which flow in and out a procedure e.g. Henry and Kafura's metric
  - counts of operands within a code procedure e.g. Halstead's metric
- cross reference and information reported therein

6.5.2 Dynamic analysis

The SMSE should support dynamic analysis by providing:

- test data generators for two types of testing:
  - black box testing (functional technique) where there is no knowledge of the internal operation of the component of software being tested and data is generated purely from the functional specification.
  - white box testing (structural technique) where the internal operation of the software is known and the data is generated in order to exercise the code and determine what it actually does.
- data flow generators that:
  - generate data for error detection
  - analyse the variables with respect of change of values and usage
  - debugging of incorrect use of parameters
  - generate the expected result
- test drivers that:
  - execute the software using files of test data and record the output
  - provide the stub (a dummy component or object used to simulate the behaviour of real component) required in top-down testing
- regression testing that:
  - records the input and output of a test session (capture)
  - reissues prerecorded inputs (replay)
  - determines that the actual results of the current test session are the same as those previous test session (compare)
- diagnosis with:
  - a sequential trace of the execution
  - a record of changes to selected data
  - an analysis of the above information to provide more comprehensive reports
- coverage analysis including:
  - code coverage

  - system coverage
  - interface coverage
- performance analysis
- test result analysis

6.5.3 Impact analysis

The SMSE should support impact analysis.

The impact analysis tool should:

- evaluate change requests for potential impact on the system, documentation, hardware, data structure and users
- utilise stability measurements and ripple effect analysis
- develop a preliminary resource estimate and provide an accurate cost of the modification
- document the scope of the requested change and update the change request

6.5.4 Traceability

The SMSE should support traceability. The traceability toolkit should provide:

- horizontal traceability, ('is implemented by') between SCIs related to different life cycle phases (for example, links between design document and code document)
- vertical traceability, ('calls', 'uses', ...) between SCIs related to the same life cycle phases (for example, links between source code SCIs, dependency analysis)
- structural traceability, ('is composed of') between SCIs of same nature
- relational traceability, ('is described by', ...) between SCIs of different types

6.6 Quality Assurance

The SMSE should support quality assurance.

The quality assurance tool should:

- manage a link between reviews and SCIs
- track the quality evolution of SCIs through the use of quality metrics
- give the current status of the product and conformance to quality plan
- indicate the trends that can be expected to influence the future status of the product
- collect metrics about maintainability
- provide information on stability of the SCIs e.g.:
  - number of change request per SCIs
  - measures of the impact of a change to single variable definition on the rest of the program modules

6.7 Planning and Controlling maintenance

The SMSE should support planning and scheduling the maintenance process as described in section 3.3.2.

6.8 Distribution

- **Physical distribution**
  The SMSE should support Local Area Networks (LAN) and Wide Area Networks (WAN) for inter site communication facilities.
- **Logical distribution**
  The SMSE should support the logical aspect of distribution (process, data base).

6.9 Others

6.9.1 Reuse

The SMSE should support reuse of SCIs at different level of abstraction (e.g. specification, design, code, documentation ...). May unsolved research problems.

6.9.2 Reverse Engineering

The SMSE should support reverse engineering. The reverse engineering tools should be used for the identification or recovery of the software requirements and/or design.

6.9.3 Safety Critical Systems

The SMSE should provide specific support for the maintenance and the verification of safety critical software and fault tolerant software e.g. Fault Tree Analysis, Software Replaceable Unit concept, safety critical software components.

6.9.4 Environment simulator

The SMSE should provide an environment simulator. The environment simulator should enable the maintainer to model the external environment of real-time software and then simulate actual operating conditions dynamically.

6.9.5 Documentation for Maintenance

The SMSE should provide documentation for maintenance. Documentation should be produced during the development phase according to the maintainer's needs and should be completed by them during the maintenance phase to complement the existing documentation and to help their personal cognitive understanding. This document should use hypertext technology easily to browse through the documentation.

6.9.6 Training

The SMSE should provide training facilities for each tool and for the environment.

6.9.7 Knowledge Transfer

The SMSE should support knowledge transfer by recording knowledge acquisition on maintainer's expertise through a knowledge base. The maintenance knowledge should be stored and will simplify the learning task for replacement personnel.

6.10 Summary

The SMSE is an environment that should support all software maintenance tasks from identification of modification to revalidation along with software configuration management and quality assurance. The SMSE is based on an open architecture that should allow the addition of new tools, a data base that should act as a central repository, and a HCI that should be flexible and tailorable to maintenance needs. The SMSE is supported by a traceability toolkit that should be interfaced with the SCM system and the toolset.

The software maintenance process should be tailored to the organisation's needs and to the SMSE.

Chapter 7

Conclusions and Further Research

This chapter presents the conclusions of the work undertaken for the thesis, including suggestions for applicable further research.

7.1 Conclusions

The research described in this thesis has achieved the objectives outlined in section 1.2 with careful use of existing ideas to support maintenance, but without radical change in technology.

The work started with the premise that software maintenance is most expensive phase of the software life cycle, and that there is a lack of good maintenance practice as well as environments for maintenance. It is well known that software maintenance dominates the high cost of software and an effective approach to reducing the maintenance cost is to provide a software maintenance method with a software maintenance support environment.

The software maintenance method has been defined as 'software maintenance best practice' based on an analytical approach of the current software maintenance process and analysis of reported maintenance problems. This method has been addressed with an hierarchical view of the software maintenance process at three different level: organisational, managerial and technical.

The organisational level has been addressed with the view of software maintenance as product support. The strategy to adopt and the identification of the market trend for the maintenance activity has been defined. The need for the involvement of senior management and effective communication in the company has been outlined.

The management level has been addressed with the provision of a maintenance plan for the management and tools to monitor and control this activity. The management of the maintenance department has been presented in a manner, such that, if adopted by a maintenance organisation, that would increase the productivity and motivation of the maintenance staff.

The technical level has been addressed with a survey of software maintenance tasks models and a proposition for a software maintenance task model for Aerospace systems. The need for software metrics applied to maintenance has been emphasised to facilitate maintenance tasks and management control.

A survey of software maintenance tools has been presented, and it has been emphasised that there is a lack of tools and methods that support the whole software maintenance process.

Current Integrated Project Support Environments have been evaluated according to their flexibility, integration and distribution, and the conclusion is that they do not support maintenance in an efficient way.

The requirements for a Software Maintenance Support Environment has been specified using the best software maintenance practice, the analysis of software maintenance tools and the evaluation of current IPSEs.

7.2 Further Research

Points for consideration:

- This software maintenance best practice is based on an analytical approach of the software maintenance process, and should be validated via experimental approach in the work environment.
- The software maintenance best practice has been described with the different organisational, managerial and technical tasks during the software maintenance process, and therefore this model should be expressed in a more dynamic style with the information flow.
- The requirements for a Software Maintenance Support Environment should be implemented in the real world in order to build a real environment for maintenance and some requirements should be used in industry to build IPSEs that address the development and maintenance of software and can provide more maintainable software.
- A plan should be elaborated on how to advise people to use this software maintenance method and how to implement the ideas of the SMSE in a company.
- A transition plan should be elaborated for the transfer from the development phase to the maintenance phase with a plan for Maintenance, for Quality Assurance and for Software Configuration Management.
- Research should be performed to find the interconnections between the software development process and the software maintenance process and the evolutionary aspects of the software maintenance process should be analysed.
- Research should be performed to develop a process model for the overall SMSE with the description of the environment's boundary with business using the process model.
- Research should be performed on the HCI with the best method to display the necessary information required for the maintainer with the surface and architecture aspects.
- Research should be performed on how to estimate the cost of modification or of a new release of the software.

- Research should be performed on recording the knowledge gained by the maintainers during the software maintenance process. This knowledge can then be used by other maintainers working in the same area or by the new maintainers to facilitate the knowledge transfer.
- Research should be performed on applying artificial intelligence techniques to the software maintenance process with the development of an intelligent maintenance assistant that can support the whole process with more automation.

Appendix A

Software Maintenance Tools Commercially Available

A.1 Tools for Program Comprehension

A.1.1 Code Analyser

(a) ACT

Tool: Analysis of Complexity Tool (ACT)

Category: Code analyser, code visualisation

Produced/Supplied by: McCabe & associates Inc., 5501 Twin Knolls Road, suite 111, Columbia, Maryland 21045

Target Language: ADA, C, FORTRAN, PASCAL, BASIC, PL/I, ASSEMBLY (8086, 6502), COBOL

Platform: runs IBM PCs, Sun, Apollo, HP, on DEC VAX workstations under Unix, and on DEC VAX mainframes under VMS with Ultrix.

Description: ACT [165] is driven by and analyses source code, producing a graphical representation of module structure, and also calculates the McCabe cyclomatic complexity metric and generates the basis set of test paths that should be exercised for each module within the source code.

This tool has been evaluated by the STSC.

(b) **AdaMAT**

Tool: AdaMAT

Category: Static code analyser

Produced/Supplied by: Dynamic Research Corporation, Andover, MA, tel: 508-475-9090

Target Language: ADA

Platform: runs on DEC VAX, Rational

Cost: $5000-$24995

Description: AdaMAT [251] analyses Ada source code against more than 150 parameters. Parameters such as relative reliability, maintainability and portability are all measured. Lower level parameters might target specific programming practices. Other criteria measured include anomalies, modularity, independence, self-descriptiveness, simplicity and clarity.

This tool has been evaluated by the STSC.

(c) **ATVS**

Tool: ATVS (Ada Test and Verification System)

Category: Static code analyser, coverage/Frequency analysis, performance analysis

Produced/Supplied by: Janice Smith, General Research Corporation, Santa Barbara, CA, tel: 805-964-7724

Target Language: ADA

Platform: runs on DEC VAX

Description: ATVS static analysis [251] examines the branching structure of the program and identifies unreachable code, identifies logic errors such as objects assigned a value and never used, and performs audits against project-specific programming standards. ATVS dynamic analysis identifies unexecuted code and aids modification of the test data to achieve complete test coverage. A task analyser is also included.

This tool has been evaluated by the STSC.

(d) **BATTLE MAP**

Tool: BATTLE MAP

Category: Code analyser, code visualisation

Produced/Supplied by: McCabe & Associates Inc., 5501 Twin Knolls

Road, suite 111, Columbia, Maryland 21045

Target Language: ADA, C, FORTRAN, PASCAL, BASIC, PL/I, ASSEMBLY (8086, 6502), COBOL

Platform: runs on PCs under MSDOS, on HP, on DEC VAX workstations under Unix, and on DEC VAX mainframes under VMS with Ultrix.

Cost: $ 6500 for PCs, $21500 for the workstation version, and $ 29000 for a 16-user VAX mainframe version.

Description: This tool [165] displays the structure of any system or subsystem graphically, using special symbols to indicate the complexity of each piece of code in the design. Battlemap allows the user to productively reverse engineer on large existing systems by providing a comprehensive, visual understanding of the entire program structure along with its quality attributes.

(e) EDSA

Tool: EDSA (Expert Dataflow and Static Analysis)

Category: Code analyser, code visualisation

Produced/Supplied by: Array Systems Computing, 5000 Dufferin Street, Suite 200, Downsview, Ont. M3H5T5, CANADA

Target Language: ADA

Platform: runs on PCs under MSDOS, on Sun, Apollo, and DEC workstations under Unix, and on VAX mainframe under Unixand on DEC VAX mainframes under VMS with Ultrix.

Cost: $ 2450 for PCs, $3250 for Sun, Apollo, and VAXstation workstation, and $ 11000 to $22000 for VAX mainframes.

Description: This tool lets the user identify which structures he/she is interested in and then removes extraneous (intervening) code, thereby letting the user see the big picture [180]. EDSA [265] provides three kind of facilities:

- It helps to browse through code following either the control flow or data flow rather than the order in which the code happens to be written.
- It displays code with unimportant source lines elided, so that the user can get a more global view of the program.
- It provides search management to make it easier to examine all possibilities when browsing.

(f) **Flint**

Tool: Flint

Category: Static code analyser

Produced/Supplied by: Pacific-Sierra Research, Los Angeles, CA, tel: 213-820-2200

Target Language: FORTRAN

Platform: runs on DEC VAX, ABM370, and any UNIX machine.

Description: Flint [251] is a lint-like utility supporting FORTRAN programs. This includes types checking and function parameter checking not performed by the FORTRAN compiler.

(g) **FORTRAN-lint**

Tool: FORTRAN-lint

Category: Static code analyser

Produced/Supplied by: Information Processing Techniques, Inc., Palo Alto, tel: 415-494-7500

Target Language: FORTRAN

Platform: runs on DEC VAX and Data General MV

Cost: \$3550 -\$9450

Description: FORTRAN-lint [251]is a static analyser that detects coding problems similar to what lint utilities do for C programs. Among the problems detected are parameter checking and type checking of variables. Other checks include the use of variables before declaration and non-use of declared variable.

(h) **F-SCAN**

Tool: F-SCAN

Category: Cross referencer, Diagram generator, Code analyser

Produced/Supplied by: Koso Inc., 114 Sansome St, Suite 1203, San Francisco CA 94104 or International Logic Corp.

Target Language: FORTRAN

Description: This tool provides structure charts, Call/Vcalled tables, Set/Used tables, and diagrams of Common.

(i) **ISAS**

Tool: ISAS

Category: Cross-referencer, diagram generator, code analyser, source code comparison

Produced/Supplied by: Singer Dalmo Victor Division, 6365 E. Tanque Verde Road, Tucson AZ 85715 or System & Software Eng.

Target Language: FORTRAN, ASSEMBLER

Description: This tool reports and charts procedure hierarchy, data references, control flow, system structure, etc.

(j) **Lint**

Tool: Lint

Category: Static code analyser

Produced/Supplied by: UNIX vendors

Target Language: C

Platform: runs on UNIX machine

Cost: Usually included in UNIX

Description: Lint [251] detects C code features that are likely to develop into bugs, procedure non-portable code, or produce inefficient code. Lint also performs a more complete type check than the C compiler. Lint detects unreachable code segments, loop errors, and parameter checking on function calls.

(k) **Lint-PLUS**

Tool: Lint-PLUS

Category: Static code analyser

Produced/Supplied by: Information Processing Techniques, Inc., Palo Alto, CA, tel: 415-494-7500

Target Language: C

Platform: runs on DEC/VAX, DATA General Nova, and Eclipse

Cost: $3550-$9450

Description: Lint-PLUS [251] is a lint utility that provides static code analysis on C code. Lint-PLUS provides information on type checking and function parameter checking. Other metrics include the conformance to standards and portability. Lint-PLUS allows the user to vary the amount of metrics received.

(l) **LOGISCOPE**

Tool: LOGISCOPE

Category: Automated Source Code Analyser (Complexity analysis, Test coverage analysis)

Produced/Supplied by: Verilog S.A., 150 rue Vauquelin, Toulouse 31081

Target Language: ADA, FORTRAN, ASSEMBLER, PASCAL, C, MODULA 2, COBOL

Platform: DEC, IBM Mainframe, SUN workstation

Cost: Fr 100000

Description: LOGISCOPE [169, 251] visualises the internal logic structure of each module of code, as well as the structural relationships of all the modules. The results provided by the Complexity Analyser are:

- textual and quantitative: Halstead, McCabe and Mohanty metrics;
- graphic and qualitative: control graphs, call graphs, criteria graphs and Kiviat diagrams;

This tool has been evaluated by the STSC.

(m) **MALPAS**

Tool: MALPAS (Malvern Program Analysis Suite)

Category: Static code analyser

Produced/Supplied by: Rex, Thompson & Partners Limited, West Street, Farnham, Surrey, GU9 7EQ, Tel: (44) 252 711414.

Platform: VAX/VMS

Cost: Fr 150000

Description: This tool [250] is a suite of software tools for the automatic, static analysis of programs written in a variety of programming languages. Six types of analysis may be performed:

- Control Flow Analyser: examines the topological structure of the software and identifies: all possible starts and ends; unreachable code and dynamic halt; the location of loops with their entry and possible and exit point and reveals the high-level control structure of the software.
- Data Use Analyser: deals with the sequential reading and writing of data and will identify unset and unused variables and incorrectly used used variables.

- Information Flow Analyser: identifies the input variables on which each output variable depends
- Partial Programmer: decompose the software into a set of sub-programs prior to semantic analysis
- Semantic Analyser: provides formulae relating the initial and final states of the variable. The results are represented as a set of disjoint input domain conditions, together with the set of output variable result expressions for each domain
- Compliance Analyser: is a variant of the semantic analyser which compares the results of the analysis with a formal specification of what the software is expected to do.

(n) MAT

Tool: MAT(Maintainability Analysis Tool)

Category: Static code analyser

Produced/Supplied by: Science Application International, Corp., Arlington VA, tel: 703-979-5910

Target Language: FORTRAN

Platform: runs on DEC/VAX, IBM, Apollo, Prime, HP, MAC's, PC's, Sun, Unisys and others.

Description: MAT [251] is a static analyser tool for FORTRAN. MAT reads, parses and analyses each FORTRAN source module. MAT provides information on errors, transportability problems, discrepancies, and poor usages. Information such as wrong data types and wrong number of arguments are provides by MAT. MAT documents each module interface and generates textual call trees and cross-referencing lists. MAT identifies all multiply defined names, circular calling of modules, and lists all callers of a module. MAT provides maintainability statistics on each modules.

This tool has been evaluated by the STSC.

(o) PC-METRICS

Tool: PC-METRICS

Category: Code analyser, quality analyser

Produced/Supplied by: SET Laboratories, Inc., Portland OR, tel: 503-289-4758

Target Language: Ada, FORTRAN, ASSEMBLER, PASCAL, C, C++, MODULA 2, COBOL

Platform: runs on UNIX systems

Cost: $199-$8500

Description: PC-METRICS [251] computes software science and cyclomatic complexity metrics. Other measurements include module size, data frequence span, coding and standards compliance. Still other metrics include the number of unique operand (see also SMR).

This tool has been evaluated by the STSC.

(p) **RXVP80**

Tool: RXVP80

Category: Cross referencer, diagram generator, test coverage analyser , code analyser, program documentation, Reformatter

Produced/Supplied by: General Research Corp., The Software Workshop, 5383 Hoolister Avenue, Santa Barbara CA 93111, tel: 805-964-7724

Target Language: FORTRAN

Platform: runs on IBM PCs

Cost: $10000

Description: This tool [66] is an automated verification system that consists of a set of tools that assist in all phases of software development. Many program errors will be detected earlier in the software life cycle, resulting in cost savings and more reliable, easier to maintain software.

RXVP80 (commercially available since 1980) includes:

- syntactical, structural, and statistical analysis to detect inconstancies in program structure and in the use of variables
- source code instrumentation
- analysis of testing coverage
- comprehensive automatic documentation

This tool has been evaluated by the STSC.

(q) **Reftran**

Tool: Reftran

Category: Code analyser, Program documentation, Cross referencer

Produced/Supplied by: William R. DeHaan

Target Language: FORTRAN

(r) **VAX SCA**

Tool: VAX Source Code Analyser

Category: Code analyser

Produced/Supplied by: DEC

Target Language: Multiple languages

Description: This tool provides facilities such as logic tracing, data flow tracing, and consistency analysis as well as cross referencer.

A.1.2 Code Visualisation

(a) **BATTLE MAP**

Tool: BATTLE MAP

Category: Code analyser, code visualisation

Produced/Supplied by: McCabe & associates, Columbia MD, tel: 800-638-6316

Target Language: ADA, C, FORTRAN, PASCAL, BASIC, PL/I, ASSEMBLY (8086, 6502), COBOL

Platform: runs on PCs under MSDOS, on HP, on DEC VAX workstations under Unix, and on DEC VAX mainframes under VMS with Ultrix.

Cost: $ 6500 for PCs, $21500 for the workstation version, and $ 29000 for a 16-user VAX mainframe version.

Description: This tool [180] displays the structure of any system or subsystem graphically, using special symbols to indicate the complexity of each piece of code in the design.

(b) **EDSA**

Tool: EDSA (Expert Dataflow and Static Analysis)

Category: Code analyser, code visualisation

Produced/Supplied by: Array Systems Computing

Target Language: ADA

Platform: runs on PCs under MSDOS, on Sun, Apollo, and DEC workstations under Unix, and on VAX mainframe under Unixand on DEC VAX mainframes under VMS with Ultrix.

Cost: $ 2450 for PCs, $3250 for Sun, Apollo, and VAXstation workstation, and $ 11000 to $22000 for VAX mainframes.

Description: This tool [180] lets the users identify which structures are interesting and then removes extraneous (intervening) code , thereby letting the users see the big picture.

(c) **GRASP/ADA**

Tool: GRASP/ADA

Category: Code visualisation

Produced/Supplied by: James Cross II, Auburn University

Target Language: ADA

Platform: runs on Sun 4 workstation under SunOS 4.0.3 or later on X Windows 11.7 or later.DEC VAX/VMS mainframes and workstations.

Cost: $ 50 distribution fee.

Description: This tool [180] builds graphical control-structure diagrams that high-light the control paths in and among Ada tasks. It is a comprehension tool tailored to a specific language.

(d) **OBJECTIVE-C Browser**

Tool: OBJECTIVE-C Browser

Category: Code visualisation, code analyser

Produced/Supplied by: Stepstone

Target Language: C

Platform: runs on Sun 3, 4 , and 386i, HP 9000, DEC VAX, and IBM RT PC workstations using Unix.

Cost: $ 995

Description: OBJECTIVE-C Browser[180] uses a windowing approach that displays hierarchical, functional, and inheritance information about code object in C or Objective-C.

It provides 3 types of information source code:

- the contents,
- available cross-referencing data, and
- source code contents with respect to the inheritance hierarchy.

(e) **SEELA**

Tool: SEELA

Category: Code visualisation, reverse engineering, source code document generator

Produced/Supplied by: Tuval Software Industries, 520 South El Camino REal, suite 700, San Mateo, CA 94402-1720, tel: 1-800-777-9996.

Target Language: ADA, C, FORTAN, PASCAL, COBOL, PL/M,

Platform: runs on DEC VAX/VMS mainframes and workstations.

Cost: $ 2000 on VAX stations

Description: This tool [180, 98] converts (using reverse engineering) code into program design language and lets the users edit the structure chart , cut and paste to and from the code, and generate high-level documentation describing the code structure. Thus, it gives and electronic path between code and its corresponding design language.

(f) **VIFOR**

Tool: VIFOR

Category: Code visualisation

Produced/Supplied by: Software tools and technologies

Target Language: FORTRAN

Platform: runs on DEC VAX station 2000 and MicroVAX IIPSs under Ultrix and on Sun workstations under Sun News.

Cost: $ 1995

Description: This tool [180] has a graphical interface that let the users select, move, and zoom into icones representing parts of the program. As such, it gives a graphical editing capability.

A.1.3 Cross Referencer

(a) **ADPL**

Tool: ADPL

Category: Program documentation, Cross referencer

Produced/Supplied by: Advanced Computer Concepts

Target Language: PASCAL, C, FORTRAN

(b) **Autoref**

Tool: Autoref

Category: Cross referencer

Produced/Supplied by: Siegel Software Service

Target Language: ASSEMBLER, COBOL

(c) **BPA**

Tool: Basic Program Analyser

(see section A.2.2).

(d) **CICS-OLFU**

Tool: CICS-OLFU

Category: Cross referencer

Produced/Supplied by: MacKinney Systems, 2674-A South Highland Avenue, Lombard IL 60148

Target Language: Any

(e) **Dossier Browse**

Tool: Dossier Browse

Category: Program documentation, Cross referencer

Produced/Supplied by: Concept Computer

Target Language: Any

(f) **F-SCAN**

(see section A.1.1).

(g) **ISAS**

(see section A.1.1).

(h) **MAD/3000**

Tool: MAD/3000

Category: Program documentation, Cross referencer

Produced/Supplied by: Related Computer Technology, 154 S. Main, Box 523, Keller TX 76248

Target Language: COBOL, FORTRAN, BASIC

(i) **Reftran**

(see section A.1.1).

(j) **SOFTOOL Programming Environment**

(see section A.3.1).

(k) **Source Print**

(see section A.2.2).

A.1.4 Source Code Comparison

(a) **ISAS**

(see section A.1.1).

(b) **Matchbook**

Tool: Matchbook

Category: Source code comparison

Produced/Supplied by: Westinghouse Management Systems

Target Language: ASSEMBLER

(c) **S/Compare**

Tool: S/Compare

Category: Source code comparison

Produced/Supplied by: ALDON Computer Group

Target Language: C

(d) **Text Comparator**

Tool: Text Comparator

Category: source code comparison

Produced/Supplied by: Dataware

Target Language: COBOL, ASSEMBLER

A.1.5 Execution Monitoring /Debugging

(a) **C-Tracer**

Tool: C-Tracer

Category: Execution monitoring/debugging

Produced/Supplied by: IPT Corp.

Target Language: C

Description: This tool provides a history of a program's execution by building a record of various program statements as they are executed.

(b) **FBUG/1000**

Tool: FBUG/1000

Category: Execution monitoring/debugging

Produced/Supplied by: Corporate Computer Systems Inc.

Target Language: FORTRAN

(c) **Intertest/CICS**

Tool: Intertest/CICS

Category: Execution monitoring/debugging

Produced/Supplied by: On-Line Software International, Inc., Executive Drive, Fort Lee NJ 07024

Target Language: ASSEMBLER

(d) **JSADebug-Assembler**

Tool: JSADebug-Assembler

Category: Execution monitoring/debugging

Produced/Supplied by: Computer Consulting & Software

Target Language: ASSEMBLER

(e) **Superbug**

Tool: Superbug

Category: Execution monitoring/Debugging

Produced/Supplied by: Technology Consulting Corporation

Target Language: ASSEMBLER

(f) **Trace**

(see section A.3.1).

(g) **Tracer**

Tool: Tracer

Category: Execution monitoring/debugging

Produced/Supplied by: IPT Corp., Palo Alto CA, tel: 415-494-7500

Target Language: FORTRAN, ASSEMBLER

(h) **XDebug**

Tool: XDebug

Category: Execution monitoring/Debugging

Produced/Supplied by: Kolinar Corp.

Target Language: ASSEMBLER

(i) **XPF/Assembler**

(see section A.3.1).

A.2 Tools for Reverse Engineering

A.2.1 Restructurer

(a) **SPAG**

Tool: SPAG

Category: Restructurer

Produced/Supplied by: OTG System Inc., Suite 300, P.O.BOX 5250, 308 Mulberry Street, Scranton PA 18505-5250 USA.

Target Language: FORTRAN

Part of the PRISM Toolkit

A.2.2 Reformatter

(a) **BPA**

Tool: Basic Program Analyser

Category: Cross referencer, Reformatter

Produced/Supplied by: Expert Systems

Target Language: BASIC

(b) **Basic-Doc**

Tool: Basic-Doc

Category: Reformatter

Produced/Supplied by: Applied Business Systems

Target Language: BASIC

(c) **RXVP80**

(see section A.1.1)

(d) **SEELA**

(see section A.1.2).

(e) **Source Print Tool:** Source Print

Category: Cross referencer, Reformatter

Produced/Supplied by: Aldebaran Laboratories, or Powerline, Inc., 2531 Baker Street, San Francisco CA 94123 USA

Target Language: FORTRAN, COBOL, C, PASCAL, DBASE, MODULA 2

Description: This tool pages, indexes, and annotated with structure lines source code.

A.2.3 Reengineering

(a) **BAL/SRW**

Tool: Basic Assembler Language Software Re-engineering Workbench

Category: Re-Engineering

Produced/Supplied by: Daniel Marks, Andersen Consulting, 33 West Monroe Street, Chicago, Illinois 60603, tel: 312-507-6748.

Target Language: Assembly

Platform: SUN under UNIX, use of X-Windows.

Description: The BAL/SRW [133] is a set of software re-engineering tools to help an analyst to recover the design of an assembly program. This is achieved through a series of abstractions, which effectively collapse program functionality into progressively higher level concepts. The program is analysed and its internal representation is created in the knowledge base. In order to learn about the program logic the analyst can

- Search for programming patterns present in the program and replace them with natural or formal language sentences in order to make the code more understandable.
- Navigate through both the source code and control flow view of the program.
- Simplify the program automatically by recognising control flow patterns, identifying subroutines and unreachable sections of the code, and by hiding selected control flow paths upon specified conditions.
- Simplify the program manually by substituting analyst defined comments for program sections.

A.2.4 Reverse Engineering

(a) **Reverse Engineering**

Tool: Reverse Engineering

Category: reverse Engineering, Code analyser, Program documentation

Produced/Supplied by: Advanced Systems Technology Corp., 9111 Edmonston Road, suite 404, Greenbelt MD 20770 USA.

Target Language: FORTRAN, C, ASSEMBLY

Description: This tool translates source languages into a specifications language (PSL/PSA).

(b) **SEELA**

Tool: SEELA

Category: Code visualisation, reverse engineering, source code document generator

Produced/Supplied by: Tuval Software Industries, 520 South El Camino REal, suite 700, San Mateo, CA 94402-1720, tel: 1-800-777-9996.

Target Language: ADA, C, FORTAN, PASCAL, COBOL, PL/M,

Platform: runs on DEC VAX/VMS mainframes and workstations.

Cost: $ 2000 on VAX stations

Description: This tool [180] converts using reverse engineering code into program design language and lets the users edit the structure chart , cut and paste to and from the code, and generate high-level documentation describing the code structure. Thus, it gives an electronic path between code and its corresponding design language.

(c) **Software Refinery**

Tools: REFINE, DIALECT and INTERVISTA

Category: Reverse Engineering

Produced/Supplied by: Lawrence Markosian, Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, tel: 415-494-6201.

Cost: see each tool

Target Language: C, ADA, Fortran, Cobol, SQL.

Description: Software Refinery is a family of products for building automated software processing tools - Tools that take source code as input and/or produce source code as output. Software refinery includes three products: REFINE, DIALECT, and INTERSTA.

(d) **REFINE**

Tool: REFINE

Category: Re-Engineering

Produced/Supplied by: Lawrence Markosian, Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, tel: 415-494-6201.

Cost: $ 7900 on SUN-3 and $10700 on SPARC

Target Language: C, ADA, Fortran, Cobol, SQL.

Description: REFINE is a programming environment for building software analysis and transformation tools. Its features a very high level executable specification language, a specification language compiler, an object oriented database, a customised editor interface, and tracing and debugging tools.

(e) **DIALECT**

Tools: DIALECT

Category: Re-Engineering

Produced/Supplied by: Lawrence Markosian, Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, tel: 415-494-6201.

Cost: $ 3700 on SUN-3 and $4900 on SPARC

Target Language: C, ADA, Fortran, Cobol, SQL.

Description: DIALECT is a tool that generates programming language parsers and printers from grammars. It includes a high level language for specifying language, a specifying grammars and a grammar compiler.

(f) **INTERVISTA**

Tools: INTERVISTA

Category: Reverse Engineering

Produced/Supplied by: Lawrence Markosian, Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, tel: 415-494-6201.

Cost: $ 2300 on SUN-3 and $3100 on SPARC

Target Language: C, ADA, Fortran, Cobol, SQL.

Description: INTERVISTA is a toolkit for building graphical interfaces to Software Refinery applications. It provides windows, diagrams, menus, and hypertext.

(g) **PRODOC**

Tool: PRODOC re/NuSys Workbench

Category: Reverse Engineering

Produced/Supplied by: Scandura Intelligent Systems, 1249 Greentree Lane, Narberth, PA 19072, U.S.A. (215.664.1207).

Target Language: PASCAL, C, ADA, COBOL, FORTRAN

Description: PRODOC [225] uses FLOWforms to represent systems at arbitrary levels of abstraction in a highly interactive visual environment. Among other things:

- the use of FLOWforms helps eliminate representational inconsistencies and awkward transitions between analysis and design.
- high level designs can be translated automatically into any of the languages supported by PRODOC.
- existing source code can be reverse-engineered at roughly the speed of a compiler
- PRODOC can convert from old to new environments with its ability to automatically translate between pseudocode languages.

A.3 Tools for Testing

A.3.1 Test Coverage Monitors

(a) **CCA**

Tool: CCA (Code Coverage Analyser)

Category: Test coverage monitor

Produced/Supplied by: HRB-Singer

Target Language: FORTRAN

(b) **FUS**

Tool: FUS

Category: Test coverage monitor

Produced/Supplied by: Digital Solutions

Target Language: FORTRAN

(c) **IITS**

Tool: IITS (Integrated Test Tool System)

Category: Test coverage monitor, regression testing

Produced/Supplied by: Edouard Miller, Software Research, Inc., 625 Third Street, San Fancisco CA 94107-1997

Target Language: ADA, C, FORTRAN, PASCAL, COBOL

Platform: runs on Unix, X Windows, MS-DOS, OS/2 systems.

Cost: $ 7400 for MSDOS and OS/2 systems, $ 32250 for Unix workstations.

Description: IITS [208] is a test coverage monitor and a regression testing tools. It is an utility system that can be integrated with the output of a programming environment to test and validate the results of the development efforts. The test coverage analysis tool guides development of test suites, assess testing progress, and aid error detection. The regression testing tool tests and retests candidate systems' functionality.

(d) **ISAS**

Tool: ISAS

Category: Cross referencer, diagram generator, code analyser, source code comparison

Produced/Supplied by: Singler Dalmo Victor Division, System & Software Eng.

Target Language: FORTRAN, ASSEMBLER

Description: This tool reports and charts procedure hierarchy, data references, control flow, system structures ...

(e) **LOGISCOPE**

(see section A.1.1)

(f) **RXVP80**

(see section A.1.1)

(g) **SMARTS**

Tool: SMARTS (Software Maintenance and Regression Test System)

Category: Test coverage monitor, regression test

Produced/Supplied by: Software Research Inc. 625 Third Street, San Francisco, CA 94107-1997

Platform: PC or 386 machine under MS-DOS or XENIX.

Description: This tool with EXDIFF execute, evaluate and report on thousands of tests automatically. Both interactive and batch tests are scripted in easy to maintain test files. SMARTS and EXDIFF, plus CAPBAK (which capture keystrokes), plus if necessary, a 3270 emulation back end, forms a powerful system of tools for planning, executing, logging, and analysing complex repetitive test suites.

(h) **SOFTOOL Programming Environment**

Tool: Softool Programming Environment

Category: Cross referencer, Test coverage monitor

Produced/Supplied by: Softool Corp., 340 South Kellogg Ave, Goleta CA 93117

Target Language: FORTRAN, COBOL, C

(i) **TCAT**

Tool: TCAT

Category: Test coverage monitor

Produced/Supplied by: Software Research Inc. 625 Third Street, San Francisco, CA 94107-1997

Target Language: ADA, FORTRAN, Pascal, COBOL, C

Platform: IBM-PC, Sun, AT&T, DEC/VAX, Apollo

Cost: $1400-$21500

Description: This tool uses the source code to make the test suites more complete than ever before. It measures test thoroughness in terms of logical branchs, instead of statement coverage that common profilers use [251].

This tool has been evaluated by the STSC.

(j) **Testing Instrumenters**

Tool: Testing Instrumenters

Category: Test coverage monitor

Produced/Supplied by: Softool Corp., 340 South Kellogg Ave, Goleta CA 93117

Target Language: FORTRAN, COBOL, C

(k) **Trace**

Tool: Trace

Category: Execution monitoring/debugging, test coverage monitor/monitor

Produced/Supplied by: AK Inc.

Target Language: Any

(l) **TVVT**

Tool: TVVT

Category: Test coverage monitor

Produced/Supplied by: AMG Associates

Target Language: FORTRAN, JOVIAL

(m) **XPF/Assembler**

Tool: XPF/Assembler

Category: Execution monitoring/Debugging, Test coverage monitor

Produced/Supplied by: Boole & Babbage Inc., 510 Oakmead Parkway, Sunnyvale CA 94086 or Phansophic Systems, Phansophic house, Nol, York Road, UXbridge Middlesex UK8 1RN, UK

Platform: IBM

Target Language: ASSEMBLER

A.3.2 Regression Testing

(a) AUTOMATOR qa

Tool: AUTOMATOR qa

Category: Regression Testing

Produced/Supplied by: Interactive Solution Inc., Bogota, NJ, Tel:201-488-3708

Platform: IBM-PCs

Target Language: Language independent

Cost: $5495

Description: AUTOMATOR qa [251] provides repetitive task automation. It supports regression testing by the recording of scrips and also the writing of scrips in a scripting language. Performance testing is provided by a function that records the screen, keyboard and internal lock, thus providing execution data. AUTOMATOR qa has the ability to generate random tests given an array of possible entries and combining them into new tests.

This tool has been evaluated by the STSC.

(b) **AutoTester**

Tool: AutoTester

Category: Regression Testing

Produced/Supplied by: Software Recording Corp., Dallas, TX, Tel: 214-368-1196

Platform: IBM-PCs

Target Language: Language independent

Cost: $30000 for 10 copies

Description: AutoTester [251] is a capture-replay-comparator tool that is capable of testing applications on PCs, minis, and mainframe computers. The link is through asynchronous communication. It records tests and allows easy editing and playback capabilities. Autotester supports a structure method that promotes test modularisation and documentation. Procedures (scrips) may be used over and over again to test similar functions that occur at different times during a test session.

This tool has been evaluated by the STSC.

(c) **Bloodhound**

Tool: Bloodhound

Category: Regression Testing

Produced/Supplied by: Goldbrick Software

Platform: IBM-PCs

Target Language: Language independent

Cost: $50 shareware

Description: Bloodhound [251] captures an unlimited number of keystrokes and screens in text mode. Screen images can be automatically captured whenever the screen scrolls. Screens can also be captured at arbitrary points in the user program. Tests can be run after changes to see if any regressions have occurred.

This tool has been evaluated by the STSC.

(d) **CAMOTE**

Tool: CAMOTE (Computer Aided MOdule Testing and design Environment)

Category: Regression Testing, module design and test

Produced/Supplied by: T. Dogsa, University of Maribor, Faculty of Technical Sciences, Smetanova 17, YU-62000 MARIBOR.

Platform: VAX(VMS) Version 4.5

Description: CAMOTE [71] provides unit testing, program testing, test coverage data, regression testing, decision-condition coverage monitoring, automatic driver modules source generation, automatic collection of data needed in reliability research projects.

(e) **CapBak**

Tool: CapBak

Category: Regression Testing

Produced/Supplied by: Software Research Inc.

Platform: IBM PC/XT/AT

Target Language: Language independent

Description: CapBak [251] captures keystroke sequences for automatic playback. CapBak includes screen-save capabilities, replay timing adjustments, and facilities to edit captured keysave files. Dynamic playback programming is provided by the use of IF and WHILE clauses in the keysave files.

This tool has been evaluated by the STSC.

(f) **CARBONCopy**

Tool: CARBONCopy

Category: Regression Testing

Produced/Supplied by: Clyde Digital Systems, Inc.

Platform: DEC/VAX, DEC/MicroVAX

Target Language: Language independent

Description: CARBONCopy [251] is a terminal I/O capture program. Terminal keystrokes are recorded to a file where they can be replayed, edited or printed. CARBONCopy provides regression testing support.

This tool has been evaluated by the STSC.

(g) **Check*Mate**

Tool: Check*Mate

Category: Regression Testing

Produced/Supplied by: Pilot Research Associates, Inc.

Platform: IBM-PC, DEC/VAX

Target Language: Language independent

Cost: $5750 first year

Description: Check*Mate [251] can perform individual tests of new functions using keystrokes capture or manual coding depending on the complexity of the test. By using keystroke capture, testing operations need to be performed once; then they can be replayed to test the function again.

This tool has been evaluated by the STSC.

(h) **DCATS**

Tool: DCATS

Category: Regression Testing

Produced/Supplied by: System Design and Development Corp.

Platform: IBM mainframe, HP

Target Language: Language independent

Cost: $75000

Description: DCATS [251] is a capture-replay-comparator tool. It provides a method of writing a test script and inputting the expected results in order to record scripts. This script can then be executed and the results compared to the expected results. Difference in actual and expected outcomes are reported. This tool has been evaluated by the STSC.

(i) **Evaluator**

Tool: Evaluator

Category: Regression Testing

Produced/Supplied by: Cadre

Platform: IBM-PCs

Target Language: Language independent

Description: Evaluator [251] is a capture-replay tool. It has a record mode where scripts are aurtomatically recorded. In replay or playback mode, Evaluator replays the recorded keystrokes from the recording session. Playback mode can run unattended and save the results to files. In programming mode scrips may be edited in the TEST Control Language (TCL).

This tool has been evaluated by the STSC.

(j) **IITS**

(see section A.3.1)

(k) **SMARTS**

(see section A.3.1)

(l) **TRAPS**

Tool: TRAPS

Category: Regression Testing

Produced/Supplied by: TravTech, Inc.

Platform: IBM mainframe, DEC/VAX, IBM-PCs

Target Language: Language independent

Description: TRAPS [251] is a menu-driven capture-replay tool. It allows recording, editing and replay of test scripts.

This tool has been evaluated by the STSC.

(m) **UATL**

Tool: UATL (Universal Ada Test Language)

Category: Regression testing

Produced/Supplied by: J. Ziegler, ITT Avionics, 390 Washington Avenue, Nutley, NJ 07110-3603

Target Language: ADA

Platform: MicroVAX, IBM PC/AT compatible, HP 9000/serie 300 processor

Description: The UATL provides a consistency framework for testing complex systems at all stages of the software/system development, production, and maintenance cycle. It consists of a set of Ada packages that provide the user with a complete complement of standardised reusable test function; including an interactive menu driven test manager, on-line operator control displays, real-time "closed loop", test data stimulus/response, test instrument drivers, data recording [284].

(n) **VAX DEC/Test Manager (TM)**

Tool: VAX DEC/Test Manager (TM)

Category: Regression Testing

Produced/Supplied by: DEC

Platform: DEC/VAX

Target Language: Language independent

Cost: $4800-$24000

Description: VAX/Test Manager [251] automates the regression testing of software. TM runs user-supplied tests, and the results are automatically compared to their expected results. Regression testing assures that changes have not affected the previous execution of the software. TM operates both in interactive and batch modes. It has a DEC windows interface which is consistent with other window applications, making it easy to learn.

This tool has been evaluated by the STSC.

A.4 Tools for Maintenance Management

A.4.1 Software Configuration Management

(a) CCC

Tool: CCC (Change and Configuration Control)

Category: Software Configuration Management

Produced/Supplied by: SOFTOOL/K3

Platform: VAX (VMS/ULTRIX), IBM (MVS/SP, MVS/XA, VM/CMS), SUN(UNIX), HP9000(HP-UX) ...

Interface VAX: all environment including Ada.

Description: CCC offers facilities to manage all aspects of changes to any machine readable units of information (code, executable, objects, shell scripts, documents, JCL etc.) through well defined access controls, change identification, control and audit procedures.

It provides Change tracking, change control, configuration control, access control, auditing, baseline creation, impact analysis, dependency reporting, can be used to satisfy MIL-STD requirements.

(b) **CHANGEMAN**

Tool: CHANGEMAN

Category: Software Configuration Management

Produced/Supplied by: SD-SCICON/SD Software Technology Centre

Platform: MicroVAX, VAX(VMS)

Description: CHANGEMAN is built on the Oracle relational database. It provides configuration identification, change and change request control and documentation, make facility, recording of build details, impact analysis and reporting, change control authorisation and the review process, extensive reporting facilities, configuration audit, task allocation and project security features, archiving and backup facilities, fully multi-user.

(c) CMAS

Tool: CMAS (Configuration Management Automation System)

Category: Software Configuration Management

Produced/Supplied by: BTG

Platform:

Description: CMAS is an application based on SOFTOOL's CCC. It provides document production, cross referencing and status accounting.

(d) CMF

Tool: CMF (Configuration Management Facility)

Category: Software Configuration Management

Produced/Supplied by: LOGSYS (Advances Systems Ltd)

Platform: UNIX,VAX(VMS)

Description: It combines CMT and DST, complies with MIL-STD-490 and automates the implementation of DoD-STD-2167.

CMF controls software, hardware and document changes and releases throughout the system life cycle. It coordinates traditional configuration management functions with comprehensive problem reporting and tracking, powerful release management, and flexible template/form generation and support.

CMT(Configuration Management Toolkit) is an integrated set of tools for controlling change and releases in system development, and a part of CMF. It provides configuration manager integrity and different tools: verifier, configuration control, system problem reporting, version description and build specification.

DST(Documentation Support Toolkit) is an integrated set of documentation support tools for producing and controlling documents, forms and templates.

(e) CMS

Tool: CMS (Code Management System)

Category: Software Configuration Management

Produced/Supplied by: DEC

Platform: MicroVAX, VAX(VMS)

Description: CMS is a library system that allows changes to text files to be tracked, reporting when, why and by whom modification were made. It provides library management and maintains audit trail.

It may also be integrated with other DEC/VAX products: MMS; VAX/LSE; VAX/SCA; VAX/Test.

(f) DSEE

Tool: DSEE (Domain Software Engineering Environment)

Category: Software Configuration Management

Produced/Supplied by: Apollo Computer UK

Platform: Platform independent (Apollo and other workstations, PCs, minis, mainframes, embedded microprocessor systems).

Description: DSEE is one of the most sophisticated configuration tools based on Unix. DSEE [138, 139] is a set of 4 management tools: history manager, configuration manager, task manager and monitor manager. It provides storage, control and tracking of source code, concurrency control, audit trail maintenance, system building, software release, automatic change notification and support distributed environments.

(g) **ENDEVOR**

Tool: ENDEVOR(ENvironment for DEVelopment OpeRations)

Category: Software Configuration Management

Produced/Supplied by: BST(Business Software Technology)

Platform: IBM

Description: It provides inventory and library management, change control, configuration management and release management.

(h) **ISPW**

Tool: ISPW

Category: Software Configuration Management

Produced/Supplied by: Benchmark Technologies Ltd

Platform: IBM(MVS)

Description: IPSW is an IPSE which provides project/work management, tool/technology management, change control, standards, procedures and audit compliance, source management, library management and production implementation.

(i) **LIFESPAN**

Tool: LIFESPAN

Category: Software Configuration Management

Produced/Supplied by: YARD Software Systems Ltd

Platform: VAX(VMS)

Description: It provides configuration management, change control, quality assurance and automatic notification of proposed change. It provides controlled access to a software database, enabling computerised information to be identified, arranged and re-used, easily and securely.

(j) LCS/CMF

Tool: LCS/CMF (Library Control System/Change Management Facility)

Category: Software Configuration Management

Produced/Supplied by: IBM(TSO)

Platform: VAX(VMS)

Description: LCS/CMF is composed of Panvalet(source library) and Panexec (object library).

(k) MOSAIX

Tool: MOSAIX

Category: Software Configuration Management

Produced/Supplied by: GEC Software Ltd.

Platform: VAX(VMS)

Description: MOSAIX is an automated interactive database system with configuration management, quality management. It guarantees consistency among components and product composition definition.

(l) PCMS

Tool: PCMS(Product Configuration Management System)

Category: Software Configuration Management

Produced/Supplied by: SQL System International/Alcatel Engineering Support Center

Platform: VAX(VMS,ULTRIX)

Description: PCMS is an integrated development environment which provides configuration management.

(m) PCS

Tool: PCS(Project Control System)

Category: Software Configuration Management

Produced/Supplied by: Scicon Consultancy International

Platform: VAX

Description: PCS is part of BAPSE (Bates Programming Support Environment) and provides project control, documentation control and production, design and build control of CORAL/MASCOT systems, and configuration management.

(n) **PVCS**

Tool: PVCS

Category: Software Configuration Management

Produced/Supplied by: Polytron Corp., Beaverton OR, tel: 800-547-4000 /The Software Construction Company

Platform: PCs (MS-DOS, OS/2), VAX(VMS), MAcintosh(MPW), Sun(SunOS)

Description: PVCS records revisions and provides an history of revisions.

(o) **SCCS**

Tool: SCCS (Source Code Control System)

Category: Software Configuration Management

Produced/Supplied by: Supplied as part of UNIX

Platform: Various UNIX

Description: SCCS [211] provides change control, version control and maintains change history.

(p) **S/COMPARE-HARMONISER**

Tool: S/COMPARE-HARMONISER

Category: Software Configuration Management

Produced/Supplied by: ALDON Computer Group, Oakland CA, tel: 415-839-3535

Platform: IBM(MVS), IBM System 38, HP3000

Description: S/COMPARE-HARMONISER identifies changes, documents changes and integrates modifications into software.

(q) **SDS**

Tool: SDS

Category: Software Configuration Management

Produced/Supplied by: Software Science Ltd.

Platform: VAX(VMS), IBM(MVS/TSO)

Description: SDS is a database tool for teams developing large systems. SDS records attributes and references to items (but does not hold actual items), assists configuring and change reports.

(r) **SIBS**

Tool: SIBS(Software Integration & Building System

Category: Software Configuration Management

Produced/Supplied by: Marconi Radar Systems

Platform: GEC 4000 (OS4000)

Description: SIBS is a system building from given components and records versions(components, tools).

(s) **SourceTools**

Tool: SourceTools

Category: Software Configuration Management

Produced/Supplied by: Real Time Product Ltd.

Platform: VAX(VMS), PDP-11(RSX/RSTS), PCs(MSDOS)

Description: SOURCECON controls access to source files, MAKE rebuilds systems, TEXCOM and SEDIT detects differences between source files and build edit scripts. SourceTools is language independent.

(t) **SVM**

Tool: SVM

Category: Software Configuration Management

Produced/Supplied by: Semantics

Platform: IBM PC

Description: SVM provides configuration management and version control.

(u) **TAGS**

Tool: TAGS (Technology for the Automatic Generation of Systems)

Category: Software Configuration Management

Produced/Supplied by: Teledyne Brown Engineering, San Diego CA, tel: 619-260-4487

Description: TAGS is a software documentation and program simulation generator and document manager.

This tool has been evaluated by the STSC (see [252])

A.4.2 Program Synthesis

(a) **MMS**

Tool: MMS (Module Management System)

Category: Program synthesis

Produced/Supplied by: DEC

Platform: MicroVAX, VAX(VMS)

(b) **Advantage Make**

Tool: Advantage Make

Category: Program synthesis

Produced/Supplied by:

Platform: PCs

Description: Make Utility

(c) **BSM-Make**

Tool: BSM-Make

Category: Program synthesis

Produced/Supplied by:

Platform: PCs

Description: Make Utility

(d) **MAKE**

Tool: MAKE

Category: Program synthesis

Produced/Supplied by: Supplied as part of UNIX

Platform: Various UNIX

Description: The Make facility is based on the UNIX operating system. A Makefile is a kind of command file and contains two different kinds of elements:

- elements that describe dependencies between building blocks
- commands that must be executed in order to make the program system.

A.4.3 Library Management

(a) **Change Man**

Tool: Change Man

Category: Library management, change management

Produced/Supplied by: SERENA Consulting/CCR Softserv

Platform: IBM(MVS, MVS/XA, TSO/ISPF)

(b) **Librarian**

Tool: Librarian

Category: library management, change control

Produced/Supplied by: ADR(Applied Data Research)

Platform: IBM(MVS/TSO,ISPF)

(c) **PolyLibrarian**

Tool: PolyLibrarian

Category: library management

Produced/Supplied by: Polytron Corp.

Platform: PCs (MS-DOS)

Description: PolyLibrarian manages object code libraries.

(d) **VMLIB**

Tool: VMLIB

Category: library management

Produced/Supplied by: Pansofic Systems

Platform: IBM (VM/CMS)

A.4.4 Change Management

(a) **Change Man**

Tool: Change Man

Category: Software Configuration Management

Produced/Supplied by: SERENA Consulting/CCR Softserv

Platform: IBM(MVS, MVS/XA, TSO/ISPF)

Description: Library management, change management

A.4.5 Change Control

(a) **Librarian**

Tool: Librarian

Category: library management and change control

Produced/Supplied by: ADR(Applied Data Research)

Platform: IBM(MVS/TSO,ISPF)

A.4.6 Version Control

(a) **TexSys**

Tool: TexSys

Category: Version Control

Produced/Supplied by:

Platform: PCs

(b) **TLIB**

Tool: TLIB

Category: Version Control

Produced/Supplied by:

Platform: PCs

(c) **TMS**

Tool: TMS (Text Management System)

Category: Version Control

Produced/Supplied by: Marconi Radar Systems/GEC Computers Ltd.

Platform: GEC 4000 (OS4000)

Description: TMS is a document version storage and access control.

A.4.7 Product Management

(a) SABLIME

Tool: SABLIME

Category: Product Administration System

Produced/Supplied by: Steve Cichinski, AT&T Bell Laboratories, 184 Liberty Corner Road, Warren, NJ 07060, Room 4N-C01, tel: 908-580-4358.

Platform: VAX line, SUN 3/4/SPARC, AT&T ..., HP 9000-300/800, IBMS, MOTOROLA 68030, PYRAMID 9825

Description: SABLIME is a comprehensive product administration system that tracks changes to a product consisting of software, hardware, firmware, and/or documents, from its origination, through maintenance, delivery, and support. Its integrated Modification Request (MR) and Configuration Management capabilities make it a unique tool for managers and product developers alike (informations from AT&T).

(b) RA-METRICS

Tool: RA-METRICS

Category: Software Metric Repository

Produced/Supplied by: Howard Rubin Associates, Inc., Winterbottom Lane, Pound Ridge, Ny 10576, tel: 914-833-3130.

Description: RA-METRICS supports all of the management reporting metrics and it reports: functional and technical quality, user satisfaction, defects counts, CASE/Tool Usage, development/maintenance history, financial history and estimation accuracy.(from advertising)

(c) SMR

Tool: Software Metric Repository

Category: Software Metric Repository

Produced/Supplied by: Denver Metrics Group, tel: 303-360-9558, USA.

Target Language: Ada, Assembler, C, C++, Fortran, Basic, Modula-2, Cobol

Description: The Software Metric Repository is a menu and mouse driven database featuring a "point and Shot" user friendly interface. The database incorporates the software metrics generated by PC-Metric (see section A.1.1)

as well as Functions Points and project data. The browse and reporting capabilities help the user to examine and analyse the raw data.
PC-Metric is a software metric generation package. It analyses the source code and computes numerous size and complexity metrics.

Appendix B

Software Maintenance Prototypes and Research Projects

B.1 Prototypes for Program Comprehension

B.1.1 Code Analyser

(a) **AEGIS**

Tool: AEGIS

Category: Code analyser, dependency analyser

Used by: Computer Sciences Corporation

Target Language: ?, used to maintain very large Navy weapons control system

Description: The method is to capture a large volume of data about the components of the software in a data base that can be queried or from which reports can be printed.

(b) **ASAP**

Tool: ASAP (Ada Static Source Code Analyser Program)

Category: Code analyser

Target Language: ADA

Description: This tool is an automated tool for static code analysis of program written in the ADA programming language.

The purpose of this analysis is to collect and store information pertaining to be analysed ADA compilation unit's size, complexity, usage of ADA language constructs and features, and static interface with other ADA compilation units.

(c) **ISMM**

Tool: ISMM: The Incremental Software Maintenance Manager

Category: Code analyser, incremental static analyser

Prototyped by: B. Ryder, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903.

Target Language: C

Description: ISMM is a prototype software tool for incremental static analysis of C programs. The goal of ISMM is to demonstrate the feasibility and praticability of using incremental static analysis to aid in the maintenance phase of the software life cycle.

ISMM consists of two modules: FREND, a front end which parses the C source code and convert it into an annotated directed graph representation of system calling structure, and BEND, a back end which performs both the incremental and exhaustive analysis [221, 222].

(d) **noname**

Tool: noname

Category: Code analyser, dependency analyser

Prototyped by: IBM

Description: This prototype combines a data base to store the program with a display "viewer" that allows a programmer to browse easily through it in many ways to accumulate information for a maintenance task [49].

B.1.2 Program Understanding

(a) **PUNS**

Category: Program comprehension

Description: PUNS (Program Understanding Support Environment) [50]gives multiple views of the program and a strategy for moving between views and exploring views in depth. It comprises two components, a repository and a user interface.

(b) SCORE/RM

Category: Program comprehension

Prototyped by: Lloy's Register of Shipping, U.K.

Description: SCORE/RM [52] provides a mechanism by which a maintainer can systematically work through the code and comprehend its purpose, produces a set f documentation to reduce future learning curves and modify the code so that it becomes easier to maintain.

(c) **noname**

Category: Program comprehension

Prototyped by: J. Sametinger, Institut fur Wirtsinformatik, University of Linz, A-4040 Linz, Austria.

Implementation: It was implemented with C++ under UNIX on Sun Workstation.

Target Language: C ++

Description: This prototype [224] helps programmers understand object-oriented software systems written in C++. It enables its users to easily browse through the system based on the relations among its classes, files and even identifiers.

B.1.3 Knowledge Based System and Maintenance Assistant

(a) EPOS

Prototype: EPOS(Expert System for Program and System Development)

Category: Expert System, Software Configuration Management

Prototyped by:

Platform:

Description: EPOS is a generic kernel environment providing a flexible infrastructure to support the evolution of production scale software system. It has four level connected by interfaces:

- an X Window user interface
- EPOS kernel tools
- programming tools and activity manager
- a configuration management system

It utilises change orientated versioning based on functional changes , and manages the software development process through knowledge based planning of tools invocations. The product model is based on semantic data model similar to the Adele product model. Smart builds are supported and it is language and method independent.

(b) **ES**

Prototype: ES

Category: Expert System

Prototyped by: F. Cross

Description: F. Cross [64] described an E.S. approach to building an information/maintenance tool for an existing target system of both hardware and software components. The purpose of tool is to help the user identify the components they seek and to automate the identification of the remaining supporting components required. The tool uses its rules rules-based knowledge and the user selections to identify the desired components and their supporting components.

(c) **SOFTM**

Prototype: SOFTM

Category: Expert System

Prototyped by: L.Pau and J.M. Negret

Description: L.Pau and J.M. Negret [190] described a software maintenance knowledge based system called SOFTM which was designed for the following purposes:

- to assist software programmers in the application code maintenance task.
- to generate and update automatically software correction documentation.
- to help the end user register, and possibly interpret, errors in successive application code versions.

SOFTM relies on an unique ATN (Augmented Transition Network) based code description, a diagnostic inference procedure based on pattern classification, and on a maintenance log report generator. The system is able to a range of programming languages provided that code descriptors can be extracted from the code. SOFTM has 3 types of knowledge base:

- Facts about error types, error locations, diagnostic classes, and the environment.
- Code independent rules that apply to the general software maintenance task.
- Symbolic descriptors derived by rewriting, in predicate form, features of programming languages provided by the compiler, the specification language, or the data flow model.

(d) **Maintainer's Assistant**

Prototype: Maintainer's Assistant

Category: Expert System

Prototyped by: University of Durham

Description: Calliss, Kalil, Munro and Ward [39] describe an intelligent, knowledge approach to software maintenance by describing a tool that is intended to help reduce the amount of time spent analysing code. They have identified 3 types of knowledge:

- Maintenance Knowledge which is the knowledge about how the maintenance programmers do their work and is elicited from expert maintainers. This knowledge provides the bulk of a systems heuristic knowledge that dedicate the weighting patterns on searches through the expert system.
- Program Plans divided into two different categories:

  - General program plans: a small set of plans that show commonly occurring activities in computer programs.

  - Program class knowledge: a set of plans common to a particular type of program.
- Program Specific Knowledge which is the internal representation of the source code together with knowledge obtained from using static code analysis tools such as cross referencers, data flow analysers, call graph gener-

ator, etc.

(e) **MACS**

Prototype: MACS (MAintenance Assistance Capability for Software)

Category: Maintenance Assistant

Prototyped by: ESPRIT Project

Description: The aim of this project [88] is to provide assistance to maintainers in maintaining medium to large scale of software applications.

The project is based on the fact that all the basic maintenance activities require an understanding on the system. MACS presents two views of the system.

- a WHAT to describes the elements of the system
- a WHY to describe the design

MACS will also, with the exploitation of the tools being developed, guide the maintainer (HOW) to do it. Using knowledge based techniques, MACS will develop a tool kit that will allow the user to analyse an existing system, and capture information.

MACS is being designed so that it will be applicable to both new and existing applications. The tool set will be customise for the domain of the software. The initial tool set will address the C programming language, and graphic interface software. These will be adapted to exploit HOOD software development method documentation and data structures. Validation activities will take place to verify the adaptability to other domains such as COBOL.

(f) **MARVEL**

Prototype: MARVEL

Category: Maintenance assistant

Prototyped by:

Platform:

Description: MARVEL [120, 121, 122] is an intelligent assistant software engineering environment that has a certain understanding of systems being developed and how to use tools to produce software. its key feature is opportunistic processing which means that MARVEL can undertake simple development task automatically (it can detect when source modules change and initiate the appropriate drivers to rederive objects).

(g) **The Maintenance Assistant**

Prototype: The Maintenance Assistant

Category: Maintenance assistant, dependencies analysis, reverse engineering, program change analysis

Prototyped by: Norman Wilde, Department of Computer Science, Bldg. 79, University of West Florida, 11000 University Parkway, Pensacola, Florida 32514. or Software Engineering Research Center

Target Language: C

Description: The aim of this project [273]is to develop methodologies and tools to aid in the complex tasks associated with making changes to software systems. Three broad approaches are currently being explored:

- dependencies analysis which involves capturing the dependencies between entities in the software system and the development of tools to present and analyse these dependencies.
- reverse engineering which involves the identification or "recovery" of program requirements and/or design specification that can help in understanding and and modifying it.
- program change analysis which involves methods for analysing differences between two versions of a program in order to understand a change that has been made and detect possible maintenance induced errors.

(h) **Nomame**

Project: Noname

Category: Data flow analyser , maintenance assistant

Project by: Norman Wilde, Software Engineering Research Center

Description: On going effort [273] to develop strategies based on incremental data flow analysis techniques that will:

- support management by providing information that can be used to guide in the allocation of resources for testing and and other maintenance activities
- improve the effectiveness of testers by helping them to generate new tests or select regression tests that will have a high likelyhood of detecting errors and

- help programmers understand rapidly the consequences of change and thus avoid making unexpected errors.

The Software Engineering Research Center is an Industry/University research center, and the companies provide funding of $30000 per year to the center. The objective of this project is not to produce polished commercial software tool but rather to explore and test methodologies.

(i) **REDO**

Prototype: REDO (Maintenance, Validation, and Documentation of Software Systems)

Category: Software Maintenance Environment

Prototyped by: ESPRIT Project

Description: The aim of the REDO project [205] is to assist software engineers in the maintenance, restructuring and validation of large software systems, and their transportation between different environments.

The project will provide a framework around which the engineers can work and this will include both methods and tools. The approach will cover a broad range of computer science based disciplines, from formal software design methods, to artificial intelligence techniques. The work will be structured into program definition, domain specific prototype applications, research into maintenance and validation, the application of knowledge bases, toolkit construction, and integration and evaluation.

After 18 months two approaches to reverse enginnering have taken place

- the first relies on SQL database repository holding the data required for the reverse engineering process.
- the second relies on fine grain object oriented repository with associated schema descriptions

An intermediate language has been designed to connect with business application languages. The user interface is regarded as having great importance.

B.2 Prototypes for Fault Localisation

B.2.1 Fault Detection

(a) **Metric classification tree**

Prototype: Software metric classification tree help guide the maintenance of large scale systems

Category: Fault detection, Fault localisation

Prototyped by: Department of Information and Computer Science, University of California, Irvine, California 92717

Platform: The classification tree generation tools are environment independent.

Description: This study [230] proposes an automated method for generating empirically-based models of error-prone software object- These models are intended to help localise the "troublesome 20 percent" (the "80:20 rule" states that approximately 20 % of a software system is responsible for 80% of its errors). The proposed method uses a recursive algorithm to automatically generate classification trees, whose nodes are multi-valued functions based on software metrics. The proposed of the classification trees is to identify components that are likely to be error prone or costly, so that developers can focus their resources accordingly.

Feasibility study:

- 1st: 16 NASA projects (3000-112000 lines), (results 79,3% of the software modules had high development effort or faults)
- 2nd Hughes maintenance environment to identify fault prone and change prone components in a large scale system (more than 100000 lines).

(b) **New fault detection technique**

Prototype: Rethinking the taxonomy of Fault Detection techniques

Category: Fault detection

Paper from: M.Young, Depart. of Information and Computer Science, University of California, Irvine 92717

Description: The conventional classification of software fault detection techniques by their operational characteristics (static vs. dynamic analysis) is inadequate [283] as a basis for identifying useful relationship between techniques. A more useful distinction is between techniques which sample the space of possible executions, and techniques which fold the space.

B.2.2 Fault/Error Localisation

(a) **PELAS**

Prototype: PELAS (Program Error-Locating Assistant System)

Category: Error localisation

Prototyped by: Department of computer Science, Wayne State University, Detroit, MI48202

Target language: Pascal

Description: This prototype [131, 132] is an error localisation assistant system which guides a programmer during debugging of Pascal programs. The system is interactive: it queries the programmer for the correctness of the program behaviour and uses answers to focus the programer's attention on an erroneous part of the program (it can localise a faulty statement). This system uses the knowledge of program structure represented by the dependence network used by the error locating reasoning mechanism to guide the construction, evaluation and modification of hypothesis of possible causes of the errors.

(b) **POLYLITH**

Prototype: POLYLITH

Category: Module fault localisation

Description: A fault localisation capability has been incorporate into POLYLITH [102], an environment that supports the interconnection of heterogeneous (multi-language and possibly distributed) software modules. This capability originated from techniques developed in the context of diagnosis in general technical systems, and requires a knowledge base that describes both the structure and intended behaviour of the system to be diagnosed.

The POLYLITH module interconnection language (MIL) provides the description of software interconnectivity (structure), which is enhanced in the approach by attributes specifying the high level behaviour of the modules.

Furthermore, the POLYLITH software bus gives us transparent instrumentation as the actual behaviour of the system under consideration. With this information, it is possible to determine a module or set of modules, that must be faulty in order to explain the given observations.

(c) **PROUST**

Prototype: PROUST

Prototyped by: Johnson and Soloway

Category: Fault localisation

Description: PROUST [118] is a knowledge-based fault localisation system designed to create a framework sufficient to catch all possible errors in small programs. They also wanted the program to understand the nature of the bugs, state it, and suggest a form of solution. To accomplish these objectives, the system requires that the program be totally and correctly specified. The major limitations of this system is that it is extremely difficult to form such specifications even for small programs, and there is no way to guarantee the specifications are corrects even after they have been stated.

(d) **PTA**

Prototype: PTA

Category: Fault localisation

Description: PTA [42] is a Knowledge-Based Program Testing Assistant. As programs are developed and tested , a user can request that the system automatically store the test cases for future use. When a bug arises in feature being tested , the system in coordination with the user can request that the appropriate saved test cases be rerun automatically -either before the system has been repaired to aid in identifying the problem or after the system has been repaired to ensure its correctness. In conjunction with this capability, the PTA heuristically modifies the corresponding test cases when the source code is changed. This preserves the ability of the system to continue to use, if possible, previous test cases to perform a type of automated regression testing of the code .

(e) **Error localisation**

Study: Error localisation during software maintenance: generating hierarchical system description from source code alone

Category: Fault localisation, data bindings (measure of software interaction)
Study from: R.Selby and V.Basili, Depart. of Information and Computer Science, University of California, Irvine 92717 and University of Maryland
Description: The purpose of this study [229] is to quantify ratios of coupling among components and cohesion within them, and use them in the generation of hierarchical system descriptions. The ability of the hierarchical descriptions to localise errors by identifying error prone system structure is evaluated using actual errors data. An analysis of variance model is used to characterise subsystems and individual routines that had either many/few errors or high/low error correction effort.

B.3 Impact Analysis

B.3.1 Dependency Analysis

(a) **Dependency Analysis tool Set**

Category: Dependency analysis

Prototyped by: Norman Wilde, University of West Florida

Target Language: C

Platform: runs on MSDOS PCs with 2 Mbytes of RAM and Unix-based workstation.

Description: This tool is a dependency analyser and a tool for building comprehension tools. The intent behind the tool is to provide a basis for determining program dependencies (data, calling, functional and definitional), so by creating your own application specific front-end, you can tailor-make your own comprehension aid.

(b) **Intermodular Dependency**

Category: Dependency analysis

Work by: Department of Informatica e Sistemistica, University of Naples, Via Claudio, 21 80125 Napoli, Italy.

Target Language: Pascal

This paper outlines that actual and mainly potential intermodular dependencies play in the maintenance phase of a software product. The problem is discussed with reference to Pascal systems and it shows how reverse engineering and static code analysis enable the identification of the actual and potential intermodular data flow relationship [47].

B.3.2 Ripple Effect Analyser

(a) **Surgeon's Assistant**

Prototype: Surgeon's Assistant

Category: ripple effect analyser, maintenance aid

Prototyped by: Keith Gallagher, Loyola College in Maryland and the University of Mariland at Baltimore.

Target Language: C

Platform: runs on Sun workstation with Sun View under SunOS Version 4.

Description: This tool slices up programs, extract pertinent information, and displays data links and related characteristics so you can track the changes and influence on targeted structures. It delivers semantic information and editing guidance to help you formulate a maintenance solution with no undetected link to unmodified code, thereby eliminating the need for regression testing.

B.4 Management Prototypes

B.4.1 Software Configuration Management

(a) ACTM

Prototype: ACTM (Advanced Configuration Management Toolset)

Category: Configuration Management Toolset

Platform: IBM

Description: ACMT [103] assists configuration management and project management activities and supports SID (IBM Systems Integration Division) 's life cycle model orientation.

(b) ACMS

Prototype: ACMS (Automated Configuration Management System)

Category: Software Configuration Management

Description: CMS [285] enhances manual techniques for project tracking and change control. It integrates the paperwork associated with configuration management with the configuration control. Configuration management procedures start when the required paperwork describing a problem, change proposal or new function is entered into the system via standard forms on the terminal. Change notices are prepared if approved are assigned to the programmer.

(c) CLEMMA

Prototype: CLEMMA

Category: Software Configuration Management

Platform: UNIX env.

Description: CLEMMA [209] implements the basic functions of identification, analysis and change control on project configurations. It manages a library of components, which is composed of an object repository and a data description repository. It utilises relational database technology, based on an extended relational model of software development in which components have an object-oriented representation. One main feature isit's exploitation of the relational database information retrieval capabilities to enable configurations to be selected on the basis on both static and dynamic aggregates.

(d) **CRUISE**

Prototype: CRUISE (Controlling Rigourously the Use of Interfaces in Software Evolution)

Category: Software Configuration Management

Description: CRUISE [254] is based on interfaces hierarchies. It consists of a representation scheme for software evolution, a MIL to express architectural design information and attributes information for identification and retrieval, a repository (the CRUISE Grid) to store design descriptions and an analytic framework to estimate the impact of changes to design description.

(e) **GDIST**

Prototype: GDIST(Global Distribution)

Category: Configuration Control System

Description: GDIST [26] is a distributed configuration control system that adds simple access to the configuration control database (e.g. RCS, SCCS, SPMS) from anywhere in the network. It also provides automatic and reliable copying of updates, and can coordinate compilation on diverse hosts via a 'global-make' command which initiates locals 'Makes'. It checks for errors, monitors and audits and notifies affected users by E-mail.

(f) **INSCAPE**

Prototype: INSCAPE

Category: IPSE, version control system

Prototyped by: D.E.Perry, AT&T Bell Laboratories, Murray Hill, NJ 97974.

Description: INSCAPE [194] is an Integrated Software Development Environment for for building large software systems by large group of developers. The version control system (INVARIANT) extends GANDALF's SVCE through the incorporation of knowledge about the semantics of module interfaces , to achieve a more flexible method of system composition than in other typed systems. It also enables INVARIANT to distinguish between parallel versions and provide a formalisation of the notions of version equivalence and compatibility to the extent of providing the system builder with the concept of plug compatibility.

(g) **IPSEN**

Prototype: IPSEN (Incremental Project Support ENvironment)

Category: Software Configuration Management

Description: IPSEN [146] is a support environment that integrates the project management, control and development activities occurring during the software life cycle. The architecture of a software system is expressed in terms of modules and module interconnections using a particular system description language. The system architecture are created and maintained by means of integrated syntax-directed editors for the system description language and the variant descriptions. Revision control is via a mechanism similar to the revision trees used in RCS, which are created and maintained using a general interactive revision editor. Configurations are built according to a given set of variant attributes and revision time stamps, or through the use of explicit variant/revision lists.

(h) **Los Alamos Hybrid Environment**

Prototype: Los Alamos Hybrid Environment

Category: Software Configuration Management

Description: Los Alamos Hybrid Environment [61] is an integrated development/configuration management system which is a Hybrid system combining features of the VMS host operating system and elements of the Softool CCC configuration management tool.

(i) **NAVE**

Prototype: NAVE (Networked, Automated Versioning Environment)

Category: Software Configuration Management

Description: NAVE [275] is an environment that supports both a diverse host machine environment and a diverse target machine environment. It's key function is to provide the disciplines of configuration identification, configuration control, status accounting and auditing, without a high degree of administrative overhead.

(j) **ODIN**

Prototype: ODIN System

Category: Software Configuration Management

Description: ODIN System [48] is an extensible object manager for software development environment, which used Make as it's conceptual starting point. It consists of:

- a specification language for describing the objects to be managed and the tools to produce them
- an object oriented request language which a user or tool can name a desired object
- an interpreter that accepts the request and produces the object

It extends the standard UNIX hierarchical file structure by the addition of user file types and operations. it deals with the information produced by software tools by invoking the appropriate set of tools needed to generate the objects that contain the data. The specification language has been designed to allow the integration of any existing tool or set of tools into the ODIN system without modification to the tools themselves, and can easily be extended to accommodate new tools. the ODIN system does not have a specific form of built-in version control, rather it considers a version control tool to be just another tool that can be specified in the ODIN specification language.

(k) **PAPICS**

Prototype: PAPICS (Product and Project Information Control System)

Category: Software Configuration Management

Platform: VMS

Description: PAPICS [67] is built on top of a VMS kernel and has access to the tools of the OS through defined interfaces. It supports configuration management and project management for a developing software system and provides archive and help facilities. PAPICS provides facilities like: automatic configuration assembly, independent further development for all version, discrete handling of numerous versions and on-line access to all versions.

(l) **PRODAT**

Prototype: PRODAT

Category: Software Configuration Management

Description: PRODAT [15] is the database component of the PROSYT software engineering environment. It provides concepts to create and manipulate versions and configurations, and for incremental archiving of these components. It uses a procedural interface to tools and a graphical interface to users instead of a query language.

(m) RCS

Prototype: RCS (Revision Control System)

Category: Software Configuration Management

Prototyped by: W. Tichy

Description: RCS [260, 261] is a widely used source code control system that assists in keeping software system consisting of many versions and configurations well organised.

(n) SERS

Prototype: SERS (Software Engineering Release System)

Category: Software Configuration Management

Prototyped by: GTE Communication Systems

Platform: IBM 3084 (UTS), VAX/VMS

Description: SERS [206] is an interactive, menu driven configuration management system and supports configuration identification, change control, status accounting and auditing of system components. Significantly, it integrates change administration with system building and demands that the change itself actually drives the system.

It ensures integrity and completeness by tracing each problem from identification to solution throughout the life cycle. It has five functional roles: task management, file management, configuration management, report management and administration management.

(o) SHAPE

Prototype: SHAPE

Category: Version Control System

Description: The Shape [160] toolkit consists of an object base for attributes software objects, a dedicates version control system and the shape program itself. SHAPE has adopted the best concepts of make, Adele and DSEE and enhanced them with full access to the object base and support of configuration rules. SHAPE offers more complete integration between source code control and configuration control through its Attributed File Store. SHAPE operates on objects in the object base rather than on UNIX file system object as in Make. Whan invoked Shape searches the object base for objects, installs them temporally as Unix files, evokes standard Unix tools on them and stores the resulting derived objects in the object base.

(p) **SIDS**

Tool: SIDS (Self-Identifying Software)

Category: Software Configuration Management

Produced by: Honeywell Bull

Implemented in: all deliverables which includes source (typical source, JCL, COBAL Copy libraries, Include Files etc.) objects and executable forms.

Description: SIDS [91] reduces problem analysis time by marking each software change with a change identifier (transmittal number) as part of the revision level information (e.g. source name, source protection notice, base data, transmittal numbe, transmittal reason).

For source code the revision information resides in the source as comments at the beginning of the module, and for objects and executable modules the revision information is prefixed by keywords for ease of identification or extraction. An automated configuration manager is used to manage the software changes and marking.

(q) **Smalltalk-80**

Prototype: Smalltalk-80 Version Manager

Category: Source Code Version Management

Description: The code and version histories are stored in a hypertext database management system. The system provides easy access to old versions of source code. Composite source code items, such as Smalltalk class can be viewed exactly as they appeared at an earlier time using a special browser, the Version Browser. Additionally two versions of the same source code item may be viewed simultaneously with their differences highlighted

B.4.2 Inverse Software Configuration Management

(a) PISCES

Prototype: PISCES (Proforma Identification SCHEME for Configurations of Existing Systems)

Category: Inverse Software Configuration Management

Prototyped by: R. Kenning, University of Durham, UK.

Description: At Durham [129], an inverse software configuration management has been identified as the process of bringing an existing software system under configuration control. PISCES is a tool under development to help the process of bringing an existing software system under configuration control. PISCES identifies and documents the configurations of an existing system.

B.4.3 Product Management

(a) SCIMM

Prototype: SCIMM (Software Change Information for Maintenance Management)

Category: Product Management

Prototyped by: S.Cooper, University of Durham, England

Platform:

Description: SCIMM [60] is a prototype system under development for storage, retrieval and analysis of software change information. SCIMM collects and stores information about requests for changes and changes made to software systems. bit also tries to capture information about the process involved in producing the change, including the diagnostics of the problem , and the design of the change. Cross referencing procedures based on a keyword system for describing a change request and its subsequent diagnosis allow searches to be made similar past changes. It also provides change metrics based on a before/after system of program complexity measurement, about individual changes and the system being maintained as a whole.

B.5 Environment Prototypes

B.5.1 Programming Environment

(a) ADELE

Prototype: ADELE

Category: Programming Environment, Software Configuration Management

Prototyped by: J.Estublier, Laboratoire de Genie Informatique (IMAG), Grenoble

Target Language: Independent

Platform: VAX/VMS, MS-DOS, UNIX

Description: ADELE [?, 79] has four main components:

- a program editor
- compiler and debugger
- a parametrised code generator
- a user interface and a program base

Components are identified by a quadruple (family name, variant id, version id, revision number). The program base is essentially a database based MIL of program information that is used to support a configuration management system.

(b) **ALS**

Prototype: ALS (Ada Language System)

Category: Programming Environment

Description: ALS [12] supports the development of large scale Ada software for real-time microprocessor-based applications.

(c) **CONMAN**

Prototype: CONMAN

Category: Programming Environment, Software Configuration Management

Description: CONMAN consist of an object base and a set of tools to help the programmer interactively construct and debug inconsistent systems. CONMAN automatically identifies and tracks 6 kinds of inconsistencies , without requiring that the user remove them immediately. It reduces the cost of rebuilding a system after source code changes through the use of smarter re-compilation, which uses link consistency to determine which modules must be rederived.

(d) **Cronus**

Prototype: Cronus Distributed Operating System

Category: Software Development Environment, Software Configuration Management

Description: Cronus establishes a SDE for a distributed and heterogeneous set of computers. Its features includes a source control system , a Bug report manager to record organise and process reports of problems, and a configuration management plan to control distribution of software to a varied set of supported hardware/software systems [24].

(e) **DARWIN**

Prototype: DARWIN

Category: Programming Environment, Software Configuration Management

Description: DARWIN supports the notion of law-governed systems and consensus based configuration binding. It views the system as a collection of attributed objects , grouped into classes to form an inheritance hierarchy. Development and system evolution is managed by the passing of a message between object according to rules, the law of the system, which define what can be done to an object. Such a framework supports consensus based configuration binding which takes into account all the constraints imposed by managers, builders and users on the use of versions.

(f) **DIF**

Prototype: DIF (Documents Integration Facility)

Category: Programming Environment, Software Configuration Management

Description: DIF [86] is a software hypertext system which when combined with several software engineering tools provides an environment for integrating and managing the document and code produced during the software life cycle. The NuMIL processing environment is used to manage the design and evolution of software configurations, the NIVEZ system represents the descriptions of configurations in a graphical manner, and RCS is used for version control.

(g) **GANDALF**

Prototype:

Category: Software Development Environment, Software Configuration Management

Description: GANDALF [179, 94] is implemented as an extension to UNIX and designed for projects that use Ada. It consists of three main components:

- an Integrated Program Construction Facility including a syntax directed editor and a syntax directed debugger
- a System Composition and Generation Facility providing a system generation facility based on system descriptions and consists of both Cooprider's version control system and Tichy's Software Development Control Facility
- a Project Management Facility dealing which issues such as conflict avoidance, access rights and documentation control.

(h) **NuMIL**

Prototype: NuMIL

Category: Programming Environment

Prototyped by:

Platform:

Description: The NuMIL environment [176] controls software development and maintenance through system descriptions stored in the INGRES relational database. The system consists of two central repositories of information: the first holds processes NuMIL descriptions, and the second consists of all the source files and revisions which are stored using RCS. It uses the notion of families to control incremental modification of systems and to provide feedback about effect of proposed changes to a system. Preconditions and postconditions are used to emphasizes behavioural aspects of a system. It also supports the notion of upward compatibility as a means of reducing the cost of analysing the effect of alterations to system configuration.

B.5.2 Software Maintenance Environment

(a) **A.S.U.**

Project: A.S.U.

Category: Software Maintenance Environment

Project by: Arizona State University

Description: The objective of this project [55] is the development of a practical software maintenance environment to support managerial and technical maintenance tasks which include:

- understanding software
- changing software
- tracing ripple effect
- retesting changed software
- documenting acquired knowledge
- planning and scheduling maintenance tasks

(b) **GALILEO**

Prototype: GALILEO

Category: Software Maintenance Environment, Software Configuration Management

Prototyped by: Rational Technology

Description: GALILEO [212] provides change control and configuration management in a distributed, heterogeneous environment. It is a client server system and is based on the Ingres relational database. It offers change control facilities similar to those of SCCS, RCS and CMS, but augments change management with methods for change distribution. The unit of change is the change record which binds together new versions of elements that results from modules changed for the same reason. It does not rigorously enforce the parallel development approach, but embodies a dynamic model of maintenance which allows maintainers to build upon each others work, taking updates from the master version before making changes. Integration testing of changes is carried out at client sites which are selected to encompass the variety of dissimilar hardware and operating systems supported.

(c) **ISCM**

Prototype: ISCM

Category: Software Maintenance Environment, Software Configuration Management

Platform:

Description: ISCM is integrated software maintenance environment for software maintenance. The essential feature is bridging configuration management and quality management. It consists of three major subsystems [7]:

- an Extended Configuration Management System (ECMS)
- a Problem Report Management and Inquiry System (PROMIS)
- a Reference Evaluator for Mode and Interface (REMIE)

which are all coordinate through a relational database management system. Configuration management is based on the property of conformity, well formedness and upward compatibility. CHILL is used to manage the resources of the system such as type definitions and global names, and maintains change histories and information and information regarding verification of changes.

(d) **MICROSCOPE**

Prototype: MICROSCOPE

Category: Software Development/Maintenance Environment, program analysis system

Prototyped by: HP Laboratories, P.O. Box 10490, Palo Alto CA 94303-0971

Description: Microscope [4] is a knowledge-based tool to assist programmers in developing an understanding of large and complex programs. This prototype provides static and dynamic analysis, execution monitoring and assistance with program modification and bug location. All program information, including source, documentation, execution histories, program analysis result and Microscope's strategies for advising the programmer, are stored in a central knowledge base [4].

Bibliography

[1] R. Arnold, B. Blum and V. Rajlich, 1989, **Bridge Technologies for Software Maintenance**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 230-231.

[2] A. Adam, J.P. Laurent, 1980, **LAURA, A System to Debug Student Programs**, *Artificial Intellingence Vol. 15,* pp 75-122.

[3] A. Alderson, M.F. Bott and M.E. Falla, 1986, **The ECLIPSE Object Management System**, *Software Engineering Journal, January,* pp 240-246.

[4] J. Ambras and V. O'Day, 1987, **MICROSCOPE: A Program Analysis System**, *Proceedings 20th International Conference on System Sciences, Hawaii,* pp 71-81.

[5] T. Anderson and P.A. Lee, 1981, **Fault Tolerance: Principle and Practice**, *Prentice-Hall,* pp 52-53.

[6] ANSI/IEEE Std 729., 1983, **Software Engineering Standards.**,

[7] M. Aoyama, Y. Hanai, and M. Suzuki, 1988, **An Integrated Software Maintenance Environment: bridging configuration management and quality management**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 40-44.

[8] R.S. Arnold and D.A. Parker, 1982, **The Dimensions of Healthy Maintenance**, *Proceedings 6th International Conference on Software Engineering,* pp 10-27.

[9] R.S Arnold, 1989, **Software Restruring**, *Proceeding of the IEEE, Vol 77, No 4,* pp 607-616.

[10] R.S. Arnold, N.F. Schneidewind, and N. Zvegintzov, 1984, **A Software Maintenance Workshop**, *Communication of the ACM, Vol 27, no 11,* pp 1120-1121, 1158.

[11] D.J. Atkinson, M.L. James, 1990, **Applications of AI for Automated Monitoring: The Sharp System**, *Proceedings AIAA/NASA.*

[12] W. Babich, 1986, **Software Configuration Management**, *Addison-Wesley,* pp 162.

[13] V.R. Basili and H.D. Mills, 1982, **Understanding and Documenting Programs**, *IEEE Transactions on Software Engineering, Vol 8, no 3,* pp 270-283.

[14] V.R. Basili, January 1990, **Viewing Maintenance as Reuse-Oriented Software Developement**, *IEEE Software,* pp 19-25.

[15] P. Baumann and D. Kohler, 1988, **Archiving Versions and Configurations in the Database System for System Engineering Environment**, *International Workshop on Software Version and Configuration Control,* pp 313-325.

[16] F.L. Bauer, 1976,, **Programming as an Evolutionary Process**, *Lecture notes in Computer Science, 46, Springer-Verlag.*

[17] F.L. Bauer, 1979, **Program Development by Stepwise Transformations the Project CIP**, *Lecture notes in Computer Science, 69, Springer-Verlag.*

[18] L. Belady and M. Lehman, 1972, **An Introduction to Growth Dynamics**, *Statistical Computer Performance Evaluation, W. Freiberger (Ed.), Academic Press,* pp 503-511.

[19] K.H. Bennett, B.J. Cornelius, M. Munro and D.J. Robson, 1988, **Software Maintenance : A Key Area For Research**, *University Computing, 10 (4),* pp 184-188.

[20] K.H. Bennett, 1989, **Software Engineering Environments : Research and Practice**, *Ellis Horwood.*

[21] K.H. Bennett, 1990, **REFORM: Transforming Code into Specifications**, *Proceedings 7 DPMA.*

[22] K.H. Bennett, 1990, **The Process of Software Maintenance**, *to be published.*

[23] K.H. Bennett, E. Younger, J. Estdale, I. Khabaza, M. Price and H. van Zuylen, 1990, **Reverse Engineering Handbook**, *2487-TN-WL-1027, Version No 0.3.*

[24] P. Bicknell, 1988, **Software Development and Configuration Management in the Cronus Distributed Operating System**, *Proceedings of Conference on Software Maintenance, IEEE, IEEE,* pp 143-151

[25] M. Branch, M. Jackson and M. Laviollete, 1985, **Software Maintenance Management**, *Proceedings Conference on Software Maintenance,* pp 62-68.

[26] P.E. Black, 1988, **GDIST: A Distributed Configuration Control System,** *International Workshop on Software Version and Configuration Control,* pp 276-284.

[27] A. Blanc and A. Mosnier, 1990, **Hermes Avionic**, *Proceedings AIAA/NASA.*

[28] J.P. Blanquart, 1990, **Ada Oriented Software Development Environment AN Example: The Hermes One**, *1st Symposium in Aerospace; Barcelona.*

[29] B.W. Boehm, 1973, **Software and its Impact: a Quantative Assessment,** *Damation, Vol 6,* pp 48-59.

[30] B.W. Boehm, E. Horowitz (Ed.), Reading, Mass: Addison-Wesley., **The High Cost of Software,** *In Pactical Strategies for Developing Large Software Systems,* 1975.

[31] B.W. Boehm, 1976, **Software Engineering,** *IEEE Transactions on Computer, 25, (12),* pp 1226-1224.

[32] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod and M.J. Merritt, 1978, **Characteristics of Software Quality**, *North-Holland Publishing Company.*

[33] B.W. Boehm, 1981, **Software Engineering Economics**, *Prentice-Hall, Englewood Cliffs, N.J.*

[34] B.W. Boehm, 1983, **The Economics of Software Maintenance**, *Proceedings Software Maintenance Workshop, IEEE,* pp 9-37.

[35] K.M. Broadley, A. Colbrook, M. Munro and D. Robson, 1989, **Block Structured Cross References for Pascal and C**, *University Computing, 11 (3),* pp 120-128.

[36] R. Brooks, 1983, **Toward a Theory of the Comprehension of Computer Programs**, *International Journal on Man-Machine Studies, 18,* pp 543-554.

[37] B. Bruegge, P. Hibbard, 1983, **Generalized Path Expressions: A High-Level Debugging Mechanism**, *The Journal of Systems and Software, Vol3* pp 265-276.

[38] F.W. Calliss, 1987, **Problems with Automatic Restructurers**, *SIGPLAN Notices, 23,* pp 13-21.

[39] F.W. Calliss, S.D. Cooper, R.J. Kenning, and M. Munro, 1988, **Notes of the Second Software Maintenance Workshop**, *Centre for Software Maintenance, Durham, England.*

[40] F.W. Calliss, M. Ward and M. Munro, 1989, **The Maintainer's Assistant**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 307-315.

[41] S. Cha, N. Leveson and T. Shimeall, 1988, **Safety Verification in Murphy Using Fault Tree Analysis**, *Proceedings 10 th Conference on Software Engineering,* pp 377-386.

[42] D. Chapman, 1982, **A Program Testing Assistant**, *Communication of the ACM,* pp 625-634.

[43] N. Chapin, 1985, **Software Maintenance: A different view**, *Proceedings of Conference AFIPS, Vol 54,* pp 509-513.

[44] N. Chapin, 1987, **The Job of Software Maintenance**, *Proceedings of Conference on Software Maintenance, IEEE, IEEE,* pp 4-12.

[45] S. Chen, K.G. Heisler, W.T. Tsai, X. Chen and E. Leung, 1990, **A Model for Assembly Program Maintenance**, *Software Maintenance: Research and Practice, Vol 2,* p.3-32.

[46] E. Chikofsky and J. Cross II, January, 1990, **Reverse Engineering and Design Recovery: a Taxomany**, *IEEE Software, Vol1, No1,* pp 13-18.

[47] A. Cimitile, G.A. Di Lucca and P. Maresca, 1990, **Maintenance and Intermodular Dependencies in Pascal Environment**, *Proceedings Conference on Software Maintenance,* pp 72-83.

[48] G.M. Clemm, 1988, **The ODIN Specification Language**, *International Workshop on Software Version and Configuration Control,* pp 144-158.

[49] L. Cleveland, 1988, **An Environment for Understanding Programs**, *Proceedings Hawaii International Conference on System Science,* pp 500-509.

[50] L. Cleveland, 1989, **An Programs Understanding Support Environment**, *IBM Syst. J. Vol 28 No 2.* pp 324-344.

[51] E.S. Cohen and all, 1988, **Version Management in Gypsy**, *ACM,* pp 201-215.

[52] A. Colbrook, C. Smythe and A. Darlison, 1990, **Data Abstraction in a Software Re-engineering Reference Model**, *Proceedings Conference on Software Maintenance*, pp 2-11.

[53] J.S. Collofello and S.J. Bortman, 1986, **An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance**, *Proceedings Phoenix Conference Computer and Communication*, *Vol 54*, pp 420-423

[54] J.S. Collofello and J.J. Buck, 1987, **Software Quality Assurance for Maintenance**, *IEEE Software*, pp 46-51.

[55] J. Collofello and M. Orn, 1988, **A Practical Software Maintenance Environment**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 45-51.

[56] M.A. Colter, 1988, **Strategies for Software Maintenance Management**, *Proceedings Conference on Software Maintenance, Chicago, Software Maintenance Association.*

[57] M.A. Colter, 1988, **The Business of Software Maintenance**, *Second Software Maintenance Workshop Notes, Centre for Software Maintenance, University of Durham.*

[58] R. Conradi, A. Lie, T.M. Didriksen, and E. Karlsson, 1989, **Change Orientated Versioning in a Software Engineering Database**, *ACM, Software Engineering Notes, Vol 14, No 7*, pp 56-65.

[59] L.L. Constantine, W.P. Stevens and G.J. Myers, 1974, **Structured Design**, *IBM Systems Journal 2*, pp 115-139.

[60] S. Cooper and M. Munro, 1989, **Software Change Information for Maintenance management**, *Technical Report 4/89, Computer Science, University of Durham.*

[61] G. Cort, 1985, **The Los Alamos Hybrid Environment: An Integrated Development/Configuration Management System**, *IEEE*, pp 11-17.

[62] E. Cougar and S. Zawacki, 1980, **Motivating and Managing Computer Personnel**, *John Wiley.*

[63] J. Collofello and L. Cousins, 1987, **Towards Automatic Software Fault Location through Decision-to-Path Analysis**, *National Computer Conference.*

[64] F. Cross, 1987, **An Expert System approach to a Program's Information/Maintenance System**, *Proceedings of Conference on Software Maintenance*

[65] B. Curtis and S.B. Sheppard, 1979, **Identification and Validation of Quantitative Measures of the Psychology Complexity of Software**, *Software Management Research.*

[66] C. Curtis and W. DeHaan, 1984, **RXVP80 - The Verification and Validation System for FORTRAN**, *Proceedings of Conference on Software Maintenance*

[67] N. Demleitner, 1988, **PAPICS: A Practical Approach to Configuration Management**, *International Workshop on Software Version and Configuration Control*, pp 381-390.

[68] J.P. Denier and J. Estublier, 1988, **Software Maintenance : a survey**, *In Proceedings International Workshop on Software Engineering and its Applications, Toulouse*, pp 323-342.

[69] V. Dhar and M. Jarke, 1988, **Dependency Directed Reasoning and Learning in Systems Maintenance Support**, *IEEE Transactions on Engineering, Vol 14, No 2*, pp 211-214.

[70] A.E. Ditri, J.C. Shaw and W. Atkins, 1971, **Managing the EDP function**, *McGraw-Hill.*

[71] T. Dogsa and I. Rozma, 1988, **CAMOTE-Computer Aided Module Testing and Design Environment**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 404-408.

[72] M. Dowson, & J. C. Wiledden, July 1985, **A Brief Report on the International Workshop on the Software Process and Software Environments**, *ACM Software Engineering Notes, Vol 10, No 3.*

[73] M. Ducassé, 1986, **OPIUM: A Sophisticated Tracting Tool for Prolog**, *Acte Seminaire de programmation en logique, CNET Lannion Tregastel.*

[74] M. Ducassé and A. Emde, 1988, **A Review of Automated Debugging Systems: Knowledge, Strategies and Technique**, *Proceedings 10 th Conference on Software Engineering*, 162-171.

[75] B. Durin, J. Abadir, B. Mouton, 1990, **Software Reuse, The Challenge of The 90's in Software Development**, *Proceedings AIAA/NASA.*

[76] M. Eisenstadt, 1985, **Retrospective Zooming, A Knowledge Based Tracing and Debugging Methodology For Logic Programming,** *Procedings of the 9th IJCAI.*

[77] ESA, 1991, **Software Engineering Standards,** *PSS-05 Issue.*

[78] ESA, 1990, **European Space Software Development Environment, Software Requirements Document,** *WME/87-409 Issue 2.*

[79] J. Estublier and N. Belkahtir, 1987, **Experiences with a Database of Programs,** *In Proceedings of the Software Engineering Symposium on Practicle Software Development Environment. ACM SIGPLAN Notices, Vol 22, No 1,* pp 84-91.

[80] M. Evans, 1989, **The Software Factory,** *John Wiley & Sons.*

[81] M. Faden, 1987, **Tools for Managing Change that Don't Mess up the System,** *DEC USER, November,* pp 44-47.

[82] R. Fairley, 1985, **Software Engineering Concepts,** *McGraw Hill, NY.*

[83] S. Feldman, 1979, **MAKE - A Computer Program for Maintaining Computer Programs,** *Software Practice and Experiences, No9.*

[84] G. Ferrand, 1987, **Error Diagnosis in Logic Programming,** *Journal of Logic Programming, No 4,* pp 177-198.

[85] S. Gamalel-Din and L. Osterweil, 1988, **New perspectives on Software Maintenance Processes,** *In Proceedings Conference on Software Maintenance.* pp 14-22.

[86] P.K. Garg and W. Scacchi, 1988, **A Software Hypertext Environment for Configured Software Descriptions,** *International Workshop on Software Version and Configuration Control,* pp 326-343.

[87] J.R. Garman, 1990, **Perspective on NASA Software Development Apollo, Shuttle, Space Station,** *Proceedings AIAA/NASA.*

[88] M. Georges, Sept 1989, **The MACS project,** *Proceedings 3rd Durham Workshop on Software Maintenance.*

[89] G.R. Gladden, 1982, **Stop the Life Cycle, I Want to Get Off,** *ACM Software Engineering Notes, 7, (2),* pp 35-39.

[90] R.L. Glass, R.A. Noiseux, 1981, **Software Maintenance Guidebook,** *Prentice-Hall.*

[91] L. Greene, 1988, **Self-Identifying Software**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 126-131.

[92] M. Griffiths, 1976, **Program Production by Successive Transformation**, *Lecture Notes in Computer Science, 46.*

[93] C. Gunn and D. Jolly, 1988, **Commercial Software - Development versus Maintenance**, *Proceedings Second Durham Workshop on Software Maintenance*

[94] N. Haberman, 1986, **GANDALF: Software Development Environment**, *IEEE Transactions on Software Engineering, SE-12,* pp 1117-1127.

[95] J. Hager, 1989, **Developing Maintainable Systems: a full life-cycle approach**, *Proceedings Conference on Software Maintenance,* pp 271-278.

[96] J. Hager, 1989, **Software Cost Reduction Methods in Practice**, *IEEE Transaction on Software Engineering, Vol 15,No 12,* pp 1638-1644.

[97] M. Harandi and J. Ning, 1988, **PAT: A Knowledge-based Program Analysis Tool**, *Proceedings Conference on Software Maintenance,* pp 312-318.

[98] J. Harband, 1990, **SEELA: Interactive Top-down Program Displays**, *Proceedings Conference on Software Maintenance,* pp 146.

[99] W. Harrison and K.I. Magel, 1981, **A Complexity Measure Based on Nesting Level**, *ACM SIGPLAN Notices,* pp 63-74.

[100] M. J. Harrold, M. Soffa and R. Gupta, 1990, **A Methodology for Controlling the Size of a Test Suite**, *Proceedings Conference on Software Maintenance,* pp 302-310.

[101] P. Hayes and J. Pepper, 1989, **Towards An Integrated Maintenance Advisor**, *Hypertext '89 Procedings,* pp 119-127.

[102] D. Hernandez and L. Kanal, 1989, **Module Fault Localisation in a Software Toolbus Based System**, *Technical Report 20742, Computer Science, University of Maryland.*

[103] C.G. Horne and R. Seeger, 1988, **An Advanced Configuration Management Tool Set**, *Proceedings Conference on Software Maintenance,* pp 229-234.

[104] R.S. Hornstein, 1990, **Distributed Decision-Making For Space Operations A Programmatic Perspective**, *Proceedings AIAA/NASA.*

[105] E. Horowitz and R. Williamson, August 1986, **SODOS: a Software Documentation Environnment : Its Definitions**, *IEEE Transaction on Software Engineering, SE-12 (8)*, pp 849-859.

[106] E. Horowitz and R. Williamson, November 1986, **SODOS: a Software Documentation Environnment : Its Use**, *IEEE Transaction on Software Engineering, SE-12 (11)*, pp 1076-1087.

[107] Hoskyns Ltd., 1973, **Implications of Using Modular Programming**, *Hoskyns Systems Research, London.*

[108] N. Howes and G. Raines, 1987, **TAVERNS and the Space Station Software Support Environment**, *Proceedings of the Ada-Europe International Conference* pp 46-58.

[109] W. Humphrey, 1989, **Managing the Software Process**, *Addison-Wesley.*

[110] ANSI/IEEE Standard 729, 1983, **IEEE Standard Glossary of Software Engineering Terminology**, *IEEE.*

[111] IEEE, 1985., **Conference on Software Maintenance-1985**, *Conference Proceedings.*

[112] IEEE, 1987, **Conference on Software Maintenance-1987**, *Conference Proceedings.*

[113] IEEE, 1988, **Conference on Software Maintenance-1988**, *Conference Proceedings.*

[114] IEEE, 1989, **Conference on Software Maintenance-1989**, *Conference Proceedings.*

[115] IEEE, 1990, **Conference on Software Maintenance-1990**, *Conference Proceedings.*

[116] W.J. Johnson, E. Soloway, 1984, **Intension-Based Diagnosis of Programming Errors**, *Proceedings of the 3rd AAAI Conference*, pp 162-168.

[117] W.L. Johnson and E. Soloway, 1985, **PROUST An Automatic Debugger for Pascal Programs**, *BYTE* pp 179-190.

[118] W.L. Johnson and E. Soloway, 1985, **PROUST Knowledge-Based Program Understanding**, *IEEE Transactions on Software Engineering, Vol 11, No 3* pp 267-275

[119] K.H. Bennett & M. Colter (Eds.), Chichester, UK., **Journal of Software Maintenance: Research and Practice**, *John Wiley Ltd*, August 1989

[120] G. Kaiser and B. Feiler, 1987, **An Architecture for Intelligent Assistance in Software Development**, *In Proceeding 9th International Conference on Software Engineering*, pp 180-188.

[121] G. Kaiser and D. Perry, 1987, **Workspaces and Experimental Databases: automated support for software maintenance and evolution**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 108-114.

[122] G. Kaiser, B. Feiler and S. Popovitch, 1990, **Intelligent Assistance for Software Developement and Maintenance**, *IEEE Software, January.*

[123] D. Kafura and G.R. Reddy, 1987, **The Use of Software Complexity Metrics in Software Maintenance**, *IEEE Transaction on Software Engineering, SE-13 (3)*, pp 303-310.

[124] V. Karakostas, 1990, **Modelling and Maintenance Software Systems at the Teleological Level**, *Software Maintenance: Research And Practice, Vol 2*, pp 47-59

[125] M. Kellner, 1988, **Modeling Software Maintenance Process: analytic summary model**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 208-285.

[126] M. Kellner and H. Rombach, 1989, **Process-Focused Model of Software Maintenance**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 81.

[127] M. Kellner, 1989, **Modeling Software Maintenance Process**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 81.

[128] M. Kellner, 1989, **Non-Traditional Perspectives on Software Maintenance**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 220.

[129] R.J. Kenning and M. Munro, 1990, **Understanding the Configuration of Operational Systems**, *Proceedings Conference on Software Maintenance*, pp 20-29

[130] B. Kernighan and P. Plauger, 1978, **The Elements of Style**, *McGraw-Hill.*

[131] B. Korel, 1988, **PELAS- Program Error-Locating Assistant System**, *IEEE Transaction on Software Engineering, SE-14 (9)*, pp 1253-1260.

[132] B. Korel and J. Laski, 1991, **Algorithmic Software Fault Localisation,** *Proceeding of the Twenty-Four Annual Hawaii International Conference on System Science.* pp 246-251.

[133] W. Kozaczynski, 1990, **Basic Assembler Language Software Reengineering Workbench,** *Proceedings Conference on Software Maintenance,* pp 215.

[134] B. Labreuille, 1990, **Overview of the Software Replaceable Unit Concept and Mechanisms Supported by the Columbus Data Management System,** *Proceedings AIAA/NASA.*

[135] D.A. Lamb, 1988, **Software Engineering: Planning for Change,** *Prentice-Hall.*

[136] P.J. Layzell and L. Macaulay, 1990, **An Investigation into Software Maintenance - Perception and Practice,** *Proceedings of Conference of Software Maintenance,* pp 130-139.

[137] R.J. Leach, 1990, **Software Metrics and Software Maintenance,** *Software Maintenance: Research and Practice, Vol 2,* pp 133-142.

[138] D.B. Leblang, 1984, **Computer Aided Software Engineering in a Distributed Workstation Environment,** *In Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment,* pp 104-112.

[139] D.B. Leblang, 1988, **Increasing Productivity with a parallel Configuration Manager,** *International Workshop on Software Version and Configuration Control,* pp 104-112.

[140] M.M. Lehman and L.A. Belady, 1976, **A Model of Large Program Development,** *IBM Syst. J., 15, (3),* pp 225-252.

[141] M.M. Lehman, 1980, **Programs, Life Cycles, and Laws of Software Evolution,** *Proceedings IEEE, 19, Vol 68, No 9,* pp 1060-1076.

[142] M.M. Lehman, 1984, **Program Evolution,** *Information Processing Management, 20,* pp 19-36.

[143] M. Lehman and L. Belady, 1985, **Program Evolution: Processes of Software Change,** *Academic Press.*

[144] P. Leluc and Y. Salomon, 1986, **Enquete sur les Couts de Maintenance,** *Genie Logiciel, Vol 5,* pp 330-366.

[145] S. Letovsky, 1986, **Cognitive Processes in Program Comprehension,** *Empirical Studies of Programmers, Ablex, Norwood,* pp 80-96.

[146] C. Lewerentz, 1988, **Variant and Revision Control within an Incremental Programming Environment,** *International Workshop on Software Version and Configuration Control,* pp 426-429.

[147] J.A. Lewis and S.M. Henry, 1989, **A Methodology for Integrating Maintainability using Software Metrics,** *Proceedings Conference on Software Maintenance,* pp 32-39

[148] J.A. Lewis and S.M. Henry, 1990, **On the Benefits and Difficulties of a Maintenability via Metrics Methodology,** *Software Maintenance: Research and Practice, Vol2,* pp 113-131.

[149] B. Lientz and E. Swanson, 1976, **The Dimensions of Maintenance,** *2nd Conference on Software Engineering.*

[150] B. Lientz and E. Swanson and G.E. Tompkins, 1978, **Characteristics of Application Software Maintenance,** *Communication of the ACM, 21, (6),* pp 466-471.

[151] B. Lientz and E. Swanson, 1980, **Software Maintenance Management,** *Addison-Wesley.*

[152] S. Linkman, L. Pickard, N. Ross, 1989, **A Pratical Procedure For Introducing Data Collection (with example from maintenance),** *In Sotware Engineering for Large Software Systems,* pp 281-303.

[153] D.C. Littman, J. Pinto, S. Levovsky and E. Soloway, 1986, **Mental Models and Software Maintenance,** *Empirical Studies of Programmers, E. Soloway and S. Iyengar, Ablex, Norwood,* pp 80-96.

[154] C.C. Liu, 1976, **A Look at Software Maintenance,** *Datamation, 22, (11),* pp 51-55.

[155] J.W. Llyod, 1986, **Declarative Error Diagnosis,** *Technical Report 86/3, Departement of Computer Science, University of Melbourne.*

[156] C. Looi, P. Ross, 1987, **Automatic Program Analysis for a Prolog Intelligent Tutoring System,** *Research Paper 307, DAI University of Eindinburgh.*

[157] F.J. Lukey, 1980, **Understanding and Debugging Programs,** *International Journal on Man-Machine Studies, Vol 12, No2* pp 75-122.

[158] F.J. Lukey, 1989, **Understanding and Debbuging Programs**, *International Journal of Man Machine Studies 12*, pp 202.

[159] J.R. Lyle and M. Weiser, 1987, **Automatic Program Bug Location by Program Slicing**, *Proceedings Second International Conference on Computers and Application.*

[160] A. Mahler and A. Lampen, 1988, **An Integrated Toolset for Engineering Software Configurations**, *Communication of the ACM*, pp 191-200.

[161] P.A. Mair, 1986, **IPSE : State of the Art Report**, *NCC Ltd.*

[162] D. Marcel, 1990, **A Study of the Impact of C++ on Software Maintenance**, *Proceedings Conference on Software Maintenance*, pp 63-69.

[163] J. Martin and C. McClure, 1983, **Software Maintenance : The Problems and its Solutions**, *Prentice-Hall.*

[164] J.A. McCall, M.A. Herdon and W.M. Osborne, 1985, **Software Maintenance Management**, *National Bureau Standards, NBS Special Publ. 500-129.*

[165] T. McCABE, 1990, **Analysis of Complexity Tool (ACT) with Battlemap (BAT)**, *Proceedings Conference on Software Maintenance*, pp 147-149.

[166] C.L. McClure, 1981, **Managing Software Development and Maintenance**, *New York : Van Nostrand.*

[167] D.D. McCracken and M.A. Jackson, 1982, **Life Cycle Concept Considered Harmful**, *ACM Software Engineering Notes 7, (2)*, pp 29-32.

[168] J.R. McKee, 1984, **Maintenance as a Function of Design**, *Proceedings of Conference AFIPS, Vol 53*, pp 187-193.

[169] J. Meekel and M. Viala, 1988, **LOGISCOPE: A Tool for Maintenance**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 328-334.

[170] J.C. Miller and B.M. Strauss III, 1987, **Implications of Automatic Restructuring of COBOL**, *SIGPLAN Notices, 22, (6)*, pp 41-49.

[171] R.J. Miara, J.A. Musselman, J A. Navarro, and B. Shneiderman, 1983, **Program Indentation and Comprehensibility**, *Communication of the ACM 26(11)*, pp 861-867.

[172] M. Munro and F.W. Calliss, 1987, **Notes of the First Software Maintenance Workshop**, *Centre for Software Maintenance, Durham, England.*

[173] W.R. Murray, 1985, **Heuristic and Formal Methods in Automatic Program Debugging**, *Proceeding of the 9th IJCAI,* pp 15-19.

[174] G.J. Myers, 1979, **The Art of Software Testing**, *New York: John Wiley.*

[175] W. Myers, 1989, **Allow Plenty of Time for Large-Scale Software**, *IEEE Software,* pp 92-99.

[176] K. Narayanaswamy. and W. Scacchi, 1987, **Maintaining Configuration of Evolving Software Systems**, *IEEE Transaction on Software Engineering, SE-13 (3),* pp 324-334.

[177] J. Neighbors, 1984, **The Draco Approach to Construction Software from Reusable Components**, *IEEE Transaction on Software Engineering, SE 10, No 5.*

[178] R.J. Martin and W.M. Osborne, 1983, **Guidance on Software Maintenance**, *National Bureau Standards, NBS Special Publ. 500-106.*

[179] D.S. Notkin, 1985, **The GANDHALF Project**, *Journal of Systems and Software, Vol 5, No 2,* PP 91-105.

[180] P. Oman, May 1990, **Maintenance Tools**, *IEEE Software,* pp 63.

[181] W. Osborne, 1987, **Building and Sustaining Software Maintainability**, *Proceedings Conference on Software Maintenance,* pp 14-23.

[182] L. Osterwiel and S. Gamalel-Din, 1988, **New Perspective on Software Maintenance Processes**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 44-22.

[183] J.C. Palous, 1990, **Columbus Data Management System**, *Proceedings AIAA/NASA.*

[184] G. Parikh, 1982, **Techniques of Program and System Maintenance**, *Winthrop Publishers.*

[185] G. Parikh, 1982, **Some Tips, Techniques, and Guidelines for Program and System Maintenance**, *Techniques of Program and System Maintenance, Winthrop Publishers,* pp 65-70.

[186] G. Parikh and N. Zvegintzov, 1983, **Tutorial on Software Maintenance**, *IEEE Computer Society.*

[187] G. Parikh, 1986, **Handbook of Software Maintenance**, *John Wiley & Sons.*

[188] D. Parnas, 1972, **On the Criteria to be used in Decomposing Systems into Modules**, *Communication of the ACM, 15(2),* pp 1053-1058.

[189] B.H. Patkaw, December 1983, **A Foundation for Software Maintenance**, *MSc. Thesis, Department of Computer Science, University of Toronto.*

[190] L. Pau and J.M. Negret, 1988, **SOFTM: A Software Maintenance Expert System in Prolog**, *Proceedings Conference on Software Maintenance,* pp 306-311.

[191] L.M. Pereira, 1986, **Rational Debugging in Logic Programming**, *3rd Logic Programming Conference, London* pp 203-210.

[192] D. Perkins, W.F. Truszkowski, 1990, **Launching AI in NASA Ground Systems**, *Proceedings AIAA/NASA.*

[193] W.E. Perry and S. Perry, 1985, **A plan of Action for Software Maintenance**, *Data Management, 23 (3).*

[194] D.E. Perry, 1987, **Version Control in the INSCAPE Environment**, *In Proceedings 9th International Conference on Software Engineering,* pp 142-149.

[195] D.E. Perry, 1989, **The INSCAPE Environment**, *Communication of the ACM,* pp 2-12.

[196] S.L. Pfleeger, 1987, **Software Engineering: the production of quality software**, *Macmillan Publishing.*

[197] S.L. Pfleeger, S.A. Bohner, 1990, **A Framework For Software Maintenance Metrics**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 320-327.

[198] R.S. Pressman, 1987, **Software Engineering: A Practitioner's Approach**, *McGraw-Hill, NY.*

[199] R.S. Pressman, 1988, **Making Software Engineering Happen**, *Prentice Hall, New Jersey.*

[200] K.J. Pulford, 1989, **The Maintenance of Large, Real-Time Embedded Systems from the Perspective of Knowledge Engineering**, *In Software Engineering For Large Software Systems,* pp 267-279.

[201] RADC-TR-86-197, 1986, **Automated Life Cycle Impact Analysis System**, *Rome Air Development Center, Air Force Systems Command, Rome, NY.*

[202] V. Rajlich, 1990, **VIFOR: Visual Interactive FORTRAN**, *Proceedings Conference on Software Maintenance.*

[203] J.L. Raney, 1990, **FY90 Status Report On SSE System Project As Viewed From Just Outside The Project**, *Proceedings AIAA/NASA.*

[204] C.V. Ramamoorthy, A. Prakash, W. Tsai and Y.Usuda, 1984, **Software Engineering: Problems and Perspectives**, *Computer, Vol. 17, No 10,* pp 191-209.

[205] T. Katsoulakis, Sept 1989, **The REDO Project**, *Proceedings 3rd. Durham Workshop on Software Maintenance.*

[206] S. Reghabi and D. Wright, 1988, **SERS: Software Engineering Release System**, *International Workshop on Software Version and Configuration Control,* pp 244-263.

[207] R.J. Reifer, 1979, **The Nature of Software Management**, *Tutorial: Software Management, IEE,* pp 2-5.

[208] S. Reisman, May 1990, **Management and Integrated Tools**, *IEEE Software,* pp 75.

[209] H.S. Render and R.H. Campbell, 1988, **CLEMMA - The Design of a Practical Configuration Librarian**, *Proceedings Conference on Software Maintenance,* pp 222-228.

[210] D.J. Robson, B.J. Cornelius and M. Munro, 1988, **An Approach to Software Maintenance Education**, *Centre for Software Maintenance, University of Durham, UK.*

[211] M. Rochkind, December, 1975, **The Source Code Control System**, *IEEE Transaction on Software Engineering,* pp 364-370.

[212] R. Rohrback and C. Seiwald, 1988, **GALILEO: A Software Maintenance Environment**, *International Workshop on Software Version and Configuration Control,* pp 265-275.

[213] H. Rombach, 1987, **A Controlled Experiment on the Impact of Software Maintenance Structure on Maintainability**, *IEEE Transaction on Software Engineering,*

[214] H. Rombach and V. Basili, 1987, **Quantitative Assessment of Software Maintenance**, *Proceedings of Conference on Software Maintenance.* pp 133-143.

[215] H. Rombach, 1988, **Can we Exploit the Relationship between Reuse and Maintenance ?, Panel Discussion**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 2-4.

[216] H. Rombach and B. Ulery, 1989, **Establishing a Measurement Based Maintenance Improvement Program: lesson learned in the SEL**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 50-59.

[217] H. Rombach, 1989, **An Experimental Process Modeling Language**, *Proceedings of Conference on Software Maintenance.*

[218] H. Rombach, 1990, **Design Measurement: some leassons learned**, *IEEE Software, March*, pp 17-24.

[219] K. Rubin, P. Jones and C. Mitchell, 1988, **A Smalltalk Implementation of an Intelligent Operator's Associate**, *Proceedings OOPSLA.*

[220] G.R. Ruth, 1976, **Intelligent Program Analysis**, *Artificial Intelligence, Vol 7, No 1*, pp 65-85.

[221] B. Ryder, 1987, **An Application of Static Program Analysis to Software Maintenance**, *Proceedings of the Twentieth Annual Hawaii International Conference on System Science*, pp 82-91.

[222] B. Ryder, 1989, **ISMM: The Incremental Software Maintenance Manager**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 142-165.

[223] G. Salton and M.J. McGill, 1983, **Introduction to Modern Information Retrieval**, *New York, NY: McGraw-Hill Book Company.*

[224] J. Sametinger, 1990, **A Tool for the Maintenance of C++ on Programs**, *Proceedings Conference on Software Maintenance*, pp 54-59.

[225] J.M. Scandura, 1990, **Cognitive Approach to Systems Engineering and Re-Engineering: Integrating New Designs with old Systems**, *Software Maintenance : Research and Practice, Vol. 2*, pp 145-156.

[226] A.J. Scheiffer, 1990, **The European Space Software Development Environment**, *Proceedings AIAA/NASA.*

[227] N.F. Schneidewind, 1987, **The State of Software Maintenance**, *IEEE Transactions on Software Engineering, 13,(3)*, pp 303-310.

[228] R.L. Seldmeyer, W.B. Thompson, P.E. Johnson, 1983, **Knowledge-based Fault Localization in Debugging**, *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging, Communication of the ACM*, pp 25-31.

[229] R. Selby and V. Basili, 1988, **Error Localization during Software Maintenance: generating hierarchical system description from source code alone**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 192-197.

[230] R. Selby and A. Porter, 1989, **Software Metric Classification Tree help guide the Maintenance of Large Scale Systems**, *Proceedings of Conference on Software Maintenance, IEEE*, pp 116-123.

[231] N. Shahmehri, M. Kamkar, and P. Fritzson, 1990, **Semi Automatic Bug Localization in Software Maintenance**, *Proceedings Conference on Software Maintenance*.

[232] D.G. Shapiro, 1981, **Sniffer: A System that Understands Bugs**, *AI Memo 638, MIT, AI Laboratory.*

[233] E.Y. Shapiro, 1982, **Algorithmic Program Debugging**, *PhD Dissertation, Yale University,* Technical report MCS8002447

[234] W.K. Sharpley, 1977, **Software Maintenance Planning for Embedded Computer Systems**, *Proceedins IEEE COMPSAC 77*, pp 520-526.

[235] S.C. Chang and C. McGowan, 1987, **Full Text Retrieval in Software Maintenance**, *Proceedings Conference on Software Maintenance*, pp 53-57.

[236] B. Shneiderman and R. Mayer, 1979, **Syntactic/Semantic Interactions in Programming Behaviour: A Model**, *International Journal on Computer and Information Science, 8(3)*, pp 219-238.

[237] B. Shneiderman, P. Shafer, R. Simon, L. Weldon, 1986, **Display Strategies for Program Browsing: Concepts and an Experiment**, *Conmputer science, Technical Report Series, University of Maryland.*

[238] A. Simon, J.L. Ducuing, O. Pasero and J.P. Denier, 1990, **Software Maintenance Ground Systems**, *International Symposium on the Management of Large Software Projects in the Space Industry*, pp 405-417.

[239] A. Simon, 1991, **Requirements for a Software Maintenance Support Environment**, *M.Sc. by Thesis, School of Engineering and Applied Science, University of Durham.*

[240] A. Simon, 1991, **Tools for Software Maintenance**, *Matra-Espace, Technical Report.*

[241] A. Simon, 1991, **Prototypes and Research Projects on Software Maintenance**, *Matra-Espace, Technical Report.*

[242] I. Smith, 1987, **Guidelines for the Maintenance and Modification of Safety Related Computer Systems**, *European Workshop on Industrial Computer Systems.*

[243] Software Maintenance News, 1986, **GSA Launches the PWB**, *September, November and December 86.*

[244] N. Zvegintzov (Ed.), 1989, **Software Maintenance Tools**, *Software Maintenance News.*

[245] H. Sneed and J. Jandrasics, 1987, **Software Recycling**, *Proceedings of Conference on Software Maintenance, IEEE, IEEE,* pp 82-90.

[246] H. Sneed, 1988, **Software Renewal: a case study**, *IEEE Software 1(3),* pp 56-63.

[247] R.A. Snowdon, North Holland., **CADES and Software Development**, *In Software Engineering Environments, H. Huenke (Ed.)* 1981.

[248] E. Soloway and W.L. Johnson, 1980, **Knowledge based program understanding**, *IEEE Transactions on Software Engineering, SE-11 (3),* pp 265-275.

[249] I. Sommerville, Second edition, 1985., **Software Engineering**, *Addison-Wesley.*

[250] DTI and NCC, 1987, **The STARTS Guide**, *NCC.*

[251] STSC, 1990, **Internal Report on Software Test Tools**, *Software Technology Support Center.*

[252] STSC, 1990, **Internal Report on Software Documentation Tools**, *Software Technology Support Center.*

[253] STSC, 1990, **Internal Report on Software Requirements Tools**, *Software Technology Support Center.*

[254] S. Subramanian, 1988, **CRUISE: Using Interface Hierarchies to Support Software Evolution**, *Proceedings of Conference on Software Maintenance, IEEE,* pp 132-142.

[255] W. Suydam, 1987, **CASE makes strides toward Automated Software Development**, *Computer Design,* pp 49-70.

[256] E.B. Swanson, 1976, **The Dimension of Maintenance**, *Proceedings 2nd International Conference of Software Engineering, IEEE,* pp 492-497.

[257] E.B. Swanson and C.M. Beath, 1990, **Departmentalization in Software Development and Maintenance**, *Communication of the ACM, Vol 33, No 6.*

[258] H. Takahashi, E. Shibayama, 1985, **PRESET - A Debugging Environment for Prolog**, *Logic Programming Conference, Tokyo,* pp 90-99.

[259] B. Terry and D. Lodgee, 1990, **Terminology for Software Engineering Environment and Computer-Aided Software Engineering**, *Software Engineering notes, Vol 15, No 2,* pp 83-94.

[260] W. Tichy, 1982, **Design, Implementation, and Evaluation of a Revision Control System**, *Proceedings of the 6th International Conference on Software Engineering,* pp 58-67.

[261] W. Tichy, 1985, **RCS - A System for Version Control**, *Software Practice and Experience, Vol 15, No 7,* pp 637-654.

[262] C. Tully (Ed.), 1989, **Proceedings of 4th. International Software Process Workshop**, *ACM Software Engineering Notes, Vol. 14, no. 4.*

[263] R.J. Turver, 1989, **Software Maintenance : Generating Front Ends for Cross Referencer Tools**, *M.Sc. by Thesis, School of Engineering and Applied Science, University of Durham.*

[264] J. Valett and F. McGarry, 1989, **A Summary of Software Measurement Experiences in the Software Engineering Laboratory**, *Journal of System and Software,* pp 136-147.

[265] L. Vanek and M. Culp, 1989, **Static Analysis of Program Source Code using EDSA**, *Proceedings Conference on Software Maintenance,* pp 192-199.

[266] Wake and J. Henry, 1988, **A Model Based on Software Quality factors which Predicts Maintainability**, *Proceeding Conference on Software Maintenance.* pp 382-387.

[267] M. Ward and M. Munro, 1988, **Intelligent Program Analysis Tools for Maintaining Software**, *The 1988 UK IT Conference, University College Swansea.*

[268] M. Ward, 1988, **Transforming a Program into a Specification**, *University of Durham, Computer Science, Technical Report 88/1.*

[269] R. Warden, 1988, **Re-Engineering for Business Change**, *Second Software Maintenance Workshop Notes, Centre for Software Maintenance, University of Durham.*

[270] H. Wertz, 1982, **Stereotyped Program Debugging**, *International Journal on Man-Machine Studies, Vol 16.*

[271] N. Wilde and S. Thebaut, 1989, **The Maintenance Assistant: work in progress**, *Journal of Systems and Software, No 9,* pp 3-17.

[272] L. Weissman, 1976, **Psychological Complexity of Computer Programs : An Experimental Methodology**, *SIGPLAN Notices, 9.*

[273] N. Wilde and S. Thebaut, 1989, **The Maintenance Assistant work in Progress**, *Journal of System and Software,* pp 3-18.

[274] A. Wingrove, 1986, **The Problem of Managing Software Projects**, *Software Engineering Journal, Vol 1,* pp 3-6.

[275] D. Wright, 1988, **Configuration Management in a Heterogeneous Environment**, *Proceedings of Conference on Software Maintenance.*

[276] S. Yau, J.S. Collofello and T. MacGregor, 1978, **Ripple Effect Analysis of Software Maintenance**, *Proceedings IEEE COMPSAC 78,* pp 492-497.

[277] S.S. Yau and J.S. Collofello, 1979, **Some stability Measures for Software Maintenance**, *Proceedings of the Computer Software and Applications Conference, IEEE, 1979* pp 674-679.

[278] S.S. Yau and J.S) Collofello, 1980, **Some Stability Measures for Software Maintenance**, *IEEE Transaction Software Engineering, Vol 6, No 6,* pp 545-552.

[279] S. Yau and J. Collofello, 1985, **Design Stability Measures for Software Maintenance**, *IEEE Transaction on Software Engineering, Vol SE-11, No 9,* pp 849-856.

[280] S. Yau and J. Tsai, 1987, **Knowledge representation of Software Component Interconnection Information for Large Scale Software Modification,** *IEEE Transaction on Software Engineering,* pp 355-361.

[281] S. Yau and S.-S. Liu, 1987, **Some Approaches to Logical Ripple Effect Analysis,** *Technical Report, SERC-TR-24-F, Software Engineering Research Center, University of Florida.*

[282] S. Yau, R. Nicholl, J. Tsai and S. Liu, 1988, **An Integrated Life Cycle Model for Software Maintenance,** *IEEE Transaction on Software Engineering. Vol SE-14, No 8,* pp 1128-1144.

[283] M. Young and R. Taylor, 1989, **Rethinking the Taxonomy of Fault Detection Techniques,** *Communication of the ACM,* pp 53-61

[284] J. Ziegler, J. Grasso and L. Burgermeister, 1989, **An Ada based Real-Time Closed-loop Integration and Regression Test Tool,** *Proceedings of Conference on Software Maintenance, IEEE,* pp 81-88.

[285] S. Zucker and K.B. Christian, 1986, **Automated Configuration Management on a DoD Satellite Ground System,** *IEEE Aerospace and Electronic Magazine,* pp 10-15.