Software Development and Continual Change: A programmers attitude problem?

Philip Andrew Harwood

Department of Computing Science and Mathematics

University of Stirling

Stirling FK9 4LA

This thesis has been submitted to the University of Stirling in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

August 1997

Abstract

Software forms around a requirement. Defining this requirement is often regarded as the hardest part of software engineering. The requirement however has an additional complexity as, once defined, it will change with time. This change of requirement can come either from the user, or from the rapid advances in 'computer' technology. How then can software succeed to continue to remain 'current' both in terms of requirements and technology in this forever changing environment?

This thesis examines the issues surrounding 'change' as applied to software and software engineering. Changing requirements are often deemed a 'curse' placed upon software engineers. It has been suggested, however, that the problems associated with change exist only in the attitude of software engineers. This is perhaps understandable considering the training methods and tools available to supposedly 'help' them.

The evidence shows that quality of management and experience of personnel involved in development contribute more significantly to the success of a development project than any technical aspect. This unfortunately means that the process is highly susceptible to staff turnover which, if uncontrolled, can lead to pending disaster for the users. This suggests a 'better' system would be developed if 'experience' was maintained at a process level, rather that at an individual level.

Conventional methods of software engineering are based upon a defined set of requirements which are determined at the beginning of the software process. This thesis presents an alternative paradigm which requires only a minimal set of requirements at the outset and actively encourages changes and additional requirements, even with a mature software product. The basis of this alternative approach is the form of the 'requirements specification' and the capturing and re-use of the 'experience' maintained by the software process itself.

Preface

This thesis is based upon a five year period of study part time at the University of Stirling. In addition, the author has full time employment as a Software Engineering Manager with a local Software House - TMS Software Systems Ltd, Alloa. It is from this base of experience that many of the issues in this document are presented and discussed.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported therein has not been presented for any university degree before, and the ideas that I do not attribute to others are due to myself.

Philip A Harwood

August 1997

Acknowledgements

Mr Charles Rattray was my supervisor in this thesis. He has to be the most motivating person I have met. There were many times during the past five years that this study was close to abandonment but each time I went to his office I would leave with a refreshing direction and desire to proceed. I appreciate all his effort and time spent with me. Thank you.

I would like to thank Dr. Robert Clark who was my 'second' supervisor, his help and comments on draft copies of this thesis are greatly appreciated.

Thom Micro Systems (TMS) have provided me with the very special opportunity of having a 'real' test bed for trying out many of the ideas presented in this thesis. I hope that the products and systems which I have developed for them over the past 11 years justify the time and effort that they have allowed me to dedicate to the work for this thesis.

The 'Elite' staff at TMS could not have been more dedicated and supportive to a manager both 'work' related and 'thesis' related. Many of them have given their own time to discuss ideas and implement solutions. I would like to give a special thank you to Andy, Stephen, Helen and Craig.

It was during a presentation to Alan Steer and Phillip LaValle of Vesuvius Co that the idea for formalising the Phase research (at that time called Foreman) was first suggested. I would like to thank them and Prof. Leslie Wilson for allowing me this opportunity.

I would like to thank Colin and Alan, two great friends who have provided overwhelming enthusiasm for this thesis, their support has been appreciated.

A special thank you is due to Paulette whom I married earlier this year. She has been full of support during these past five years and has sacrificed holidays to allow this thesis to be completed. Thanks also to Gemma and Liam for their understanding.

I would like to dedicate this thesis to my parents, Geoff and Shirley Harwood. They have always been there to support and encourage me in everything I have done. I could not wish for better parents and I hope I make them as proud of me as I am of them. Thank You.

Contents

List of Figures

Chapter 1

Introduction

It was a NATO report in 1968 [NATO68] which first identified and documented a 'software crisis'. In this report it states that, in general, software tends to be delivered over budget, over schedule and under specification. Approaching 30 years hence, current literature still reports a software crisis which relates to software being delivered over budget, over schedule and under specification. This raises the question of what advances, if any, have been made in the discipline of software engineering? This contrasts sharply with software's close companion - the hardware upon which the software operates. A 'computer' which now exists in a single chip smaller than a fingernail would have filled a jumbo jet in 1965. Advances in this technology far exceed any previous engineering discipline.

It appears that much time has been spent dealing with what Brooks calls 'Accidents' [Brooks87]. These are elements of software engineering attributed to the technology of the day and not the real 'essence' of software development. According to Brooks, the real essence of building software is the hard part : the specification, design and testing of a conceptual construct.

The existence of software was founded on, and its development continues as, a response to the demands for computer 'tools' to help with some user 'needs'. The 'deliverable' in software should be regarded as the 'satisfaction of a user need' rather than the tangible product [Cosgrove71]. The method of achieving this, according to Rowen in [Rowen90] has two major obstacles for the software developer:

"The first problem is to get unambiguous requirements from the prospective user. The second is to have a happy user when the software is delivered (exactly as specified in the requirements)."

These problems are not unique to software engineering, the same can apply to other engineering disciplines.

The Engineering Process

All engineering disciplines share a common 'theme': to create a 'solution' from the identification of a 'problem'. In civil engineering, the problem may be the need to cross a river, the solution - a bridge; in mechanical engineering, the problem may be the need for power, the solution - an engine; in electrical engineering, the problem may be the need for communication, the solution - a radio. In software engineering, the problem may be the retrieval or analysis of large amounts of information, the solution - a computer database program (software).

Each discipline has a series of activities which form together to become a 'process'. The process therefore defines the set of activities which have been proven to take a particular form of problem (or requirements) and create a solution. These activities, regardless of engineering discipline follow a pattern : the identification of the requirements specification, the design of a solution specification, the fabrication of the product to the design, the testing of the product which (if successful) leads to delivery.

The software process is a sequence of software engineering activities, performed by a 'software engineer' (the term software engineer in this text is synonymous with 'developer' or 'programmer') or a team of software engineers. The activities begin with the identification of a need (from a 'user') and concludes with the delivery of a software product (or software application) that responds effectively to the need [Blum93]. The act of creating a software product is known as 'development' or a 'project'. The effectiveness of a software process is a measure of how well these sequence of activities achieve the software product in terms of accuracy and speed of development.

Attitude to Requirements Change

In all engineering disciplines, the requirements are subject to change at any step in the process. For reasons explained later, software is perhaps more susceptible to requirements change than the rest. In addition, whilst it is recognised that changing requirements in other engineering disciplines may involve major retooling or rebuilding costs, the lack of physical items leads to a perceived ease with which software can be changed and the unwillingness to recognise the same scale of cost.

The rate at which hardware technology is progressing is far greater than that of any other engineering discipline. The performance-price gain has increased by six orders of magnitude in the past 30 years [Brooks87]. This often results in systems being redundant, or at least old-fashioned even before they are complete.

Whilst many existing software processes exist in today's technology that can be classed as effective in terms of creating a product from a defined set of requirements, there are precious few, if any, which remain effective when trying to keep the 'product' current in terms of the changing requirements. Changing of requirements is therefore a source of exasperation for software engineers and this can understandably result in a negative attitude to change.

The purpose of this thesis is to investigate why existing traditional software processes fail to remain effective when requirements change, especially after a product is 'mature'. The result of this investigation leads to an alternative paradigm and process for software engineering (or development). The basis of this paradigm is in the method of defining requirements. This leads to an abandonment of the traditional forms of a 'specification'. For reference, this process is given the name **Phase**.

Experience of Software Development

The activities which form a process are based upon the 'experience' of developing similar solutions to similar problems. Due to the fact the software engineering is only in its infancy, aged perhaps only forty years compared to the hundreds of years in civil and mechanical engineering, comparative experience in software engineering is lacking.

Experience is about being able to relate a current situation to a previous situation encountered and, knowing the outcomes of the previous situation, being able to make a more informed judgement on the action to take in the current situation. Unlike animals, humans have an ability to share experience through communication via books, speech, video etc. These methods, by their nature are a slow means of transferring knowledge. What would be ideal is the ability to 'plug in' the experience of one human directly into another and transfer all the relevant knowledge in an instant.

Whilst this thesis does not attempt any sort of physical 'wiring' of humans to transfer brain thought processes, the fact that software can be developed by using other software means that the power of the 'computer' can be used to help accelerate the 'learning' experience of software activities. A method is proposed which allows the design decision processes to be automatically recorded in such a form that it can be 're-run' in a multi-dimensional manner to give an accelerated learning experience to new software developers.

1.1 The Scope and Objective of the Thesis

This thesis is primarily concerned with investigating how software can be developed and remain 'current' in terms of satisfaction of user and technological requirements, considering that these requirements may be poorly understood and subject to continual change. To accomplish this, it was necessary to:

- assess the effectiveness of existing software processes in dealing with changing requirements;
- study the way in which requirements change and identify patterns for when and why they occur;
- analyse how the form of requirements can relate to the effectiveness of the software process;
- develop a method of capturing and specifying requirements in a form which is susceptible to changing requirements.

The result of this investigation is the **Phase** method of software development. This will be presented, describing its:

- form of specification
- set of rules for translating this specification into resultant programs
- set of heuristics .

1.2 What is Phase?

Phase is a concept which combines several 'popular' aspects of Software Engineering. Although it can be considered in several categories it is :

☒ Not just another Prototyping Tool

☒ Not just another Report Writer

☒ Not just another Automatic Code Generator

☒ Not just another Software Process

Phase is a (Program) Structure with the following attributes :

☑ The Phase Structure consists of seven simple definable Phase entities.

☑ The definition of the Phase entities specify a Phase Program Design.

☑ The Phase Design can be executed as a Phase Prototype.

☑ The Phase Prototype is a tool for extracting and refining requirements.

☑ The Phase Design entities can be used by automatic code generation routines to create programs and documentation.

☑ The Phase Process is used to effectively manage the development of Phase programs.

☑ Phase CASE tools are required to develop Phase Programs.

☑ Phase captures 'experience'.

☑ Phase programs are resilient to the detrimental effect of changes in requirements.

Figure 1.1 illustrates the principle of Phase : To provide a software development system which takes changes in requirements and maintains stable, mature software systems. This is

achieved by using a number of CASE tools which interact with the Phase Structure which represents the design of the system.

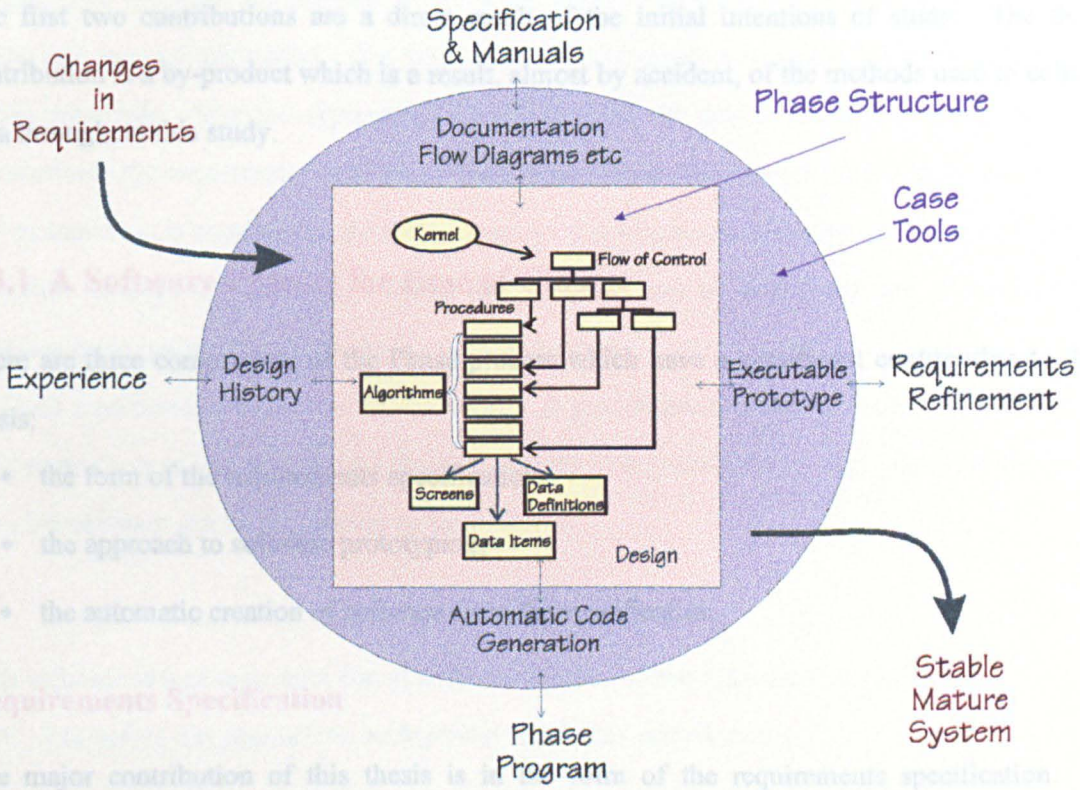

figure 1.1 : The Phase Concept

1.3 The Contribution of the Thesis

This section summarises the contributions made by the thesis :

- propose a software process which facilitates ease of change

- propose a practical method of recording design information such that the experience is reused.
- describe a generic method of monitoring and validating changes

The resultant effect is that software developed in this way is able to remain 'current' even though its requirements are changing;

The first two contributions are a direct result of the initial intentions of study. The third contribution is a by-product which is a result, almost by accident, of the methods used to collect data throughout this study.

1.3.1 A Software Process for Ease of Change

There are three components of the Phase process which have a significant contribution to this thesis:

- the form of the requirements specification;
- the approach to software prototyping;
- the automatic creation of software from the specification.

Requirements Specification

The major contribution of this thesis is in the form of the requirements specification. A requirements specification has two roles. The first is to determine the goals which will satisfy a user need. The second is to communicate these goals so that a software product can be designed to meet the original user need. Traditionally the form of the requirements in software engineering is based upon the form of requirements which is found in hardware requirements or requirements for other engineering disciplines. This is, for example, a collection of drawings, descriptions or mathematical formula. These must all be available before 'fabrication' is started. The Phase system does not preclude these forms of requirements and in fact uses some of these forms to communicate its requirements.

The Phase specification can exist only within the environment of a computer as it is a multi-dimensional repository based system. The question occurs as to whether the Phase

specification is a requirements specification (of the problem) or a design specification (of a solution)? A Phase specification is a combination of both. Simon observes that '...solving a problem simply means representing it so as to make the solution transparent' [Simon69].

Requirements which can be completely determined before fabrication can be called 'closed requirements' [Blum93]. Closed requirements are well defined and stable. There are many categories of software applications where requirements can seldom be completely determined before any form of fabrication. One such category is Interactive Business Information Systems (IBIS) especially where the application domain is relatively new to computerisation. In these applications, the requirements can be called 'open'. Open requirements are poorly understood and dynamic. It is specifically the IBIS category of applications with open requirements that is the prime concern in this study. Whilst requirements may be ill-defined, the technology for realising these types of applications is relatively mature.

Open requirements cannot be pre-specified. A specification therefore exists only in parallel with some form of system, either a finished system or a model of a system. A specification in this circumstance can be considered as 'as built'.

IBIS Software

IBIS software is a generic term for all software which has the following properties :

- Interactive (as opposed to background 'batch-job' submissions)
- Interface with human users (as opposed to electronic or mechanical process control)
- Storage, retrieval and process of similar 'sets of data' (as opposed to highly computational)
- Considered 'business critical' (as opposed to 'mission critical'). This means that failure of the software will lead to monetary loss as opposed to life loss.

IBIS software can be considered as having a structure with three main layers. This is represented in figure 1.2.

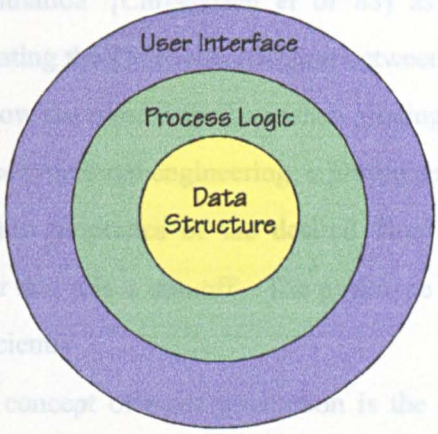

figure 1.2 : The layers of IBIS software

The layers or components of the IBIS structure are:

- the *Data Structure* is the storage of information;
- the *Process Logic* is the set of operations, which can be performed either directly on the data structure, or in transferring the information between the data structure and the user interface;
- the *User Interface* is the two-way communication of information between the software and the human operators via various forms of inputs and outputs.

Examples of IBIS software are :

- Accounting systems
- Order Processing and Stock Control systems
- Clinical Information systems
- Membership systems

This is by no means an exhaustive list.

Software Prototyping

Software prototyping in its various forms [Floyd83] has proved to be a major contributor as a method for refining requirements. The Phase process uses the rapid prototyping [Henderson86]

technique 'Front End Simulation' [Christensen *et al* 83] as a major method of refining requirements and communicating the Phase specification between developers and users.

A prototype does not follow the same definition when relating to software as it does in other engineering disciplines. In conventional engineering, a prototype is a 'first of a type'. Typically this is a product with all the properties of the desired 'final product' but which has been constructed in such a manner that it is a 'one-off'. The prototype is then examined for ways that it can be mass produced efficiently.

With software, the only concept of mass production is the duplication of the distribution media. A software prototype, in our sense, is a software model which has all the facilities of the user-interface but no process logic or data structures. The Phase software prototype is an execution of the Phase specification. It is concerned only with a subset of the specification : the user interface. As it will be shown later, for IBIS software, the user interface is seen as the key component in the specification. Whilst it can be used to determine the relationship between the 'user need' and the 'design' it is also a major contributor to the definition of the derived requirements as described in chapter 3 and in [Blum93].

A full comparison of the Phase prototyping scheme, in relation to its effectiveness in achieving the 'requirements of a prototyping scheme', is contained in chapter 8.

Automated Creation of Programs from the Specification

It is one of the major aims of the software industry to be able to automatically create executable machine instructions directly from a specification. This can be seen in the trend for higher and higher level programming languages throughout the history of software engineering. This trend can be summarised as :

- Original binary input of machine executable instructions;
- Development of assembly languages;
- Development of 'third generation' languages (3GL) and 'high level' compilers;
- Development of fourth generation languages (4GL) incorporating high level data manipulation intrinsics;
- Development of 'code generators' from formal specifications

The format of the specification in the Phase system is particularly suitable for machine automated generation into third or fourth generation programming languages.

Implementations of the Phase Process

The Phase process has been in use (although initially informally) since 1986. The requirements specification exists only within a computer repository; this insists, therefore, on a Computer Assisted Software Engineering (CASE) mechanism to maintain it.

Two implementations of a CASE system have been developed, both based on identical repository structures. There are however slight differences, described below. In the remaining text, features described and experiences reported will be set in relation to the combined features of the two systems without individual clarification. The thesis is based upon the Phase theory, not the implementation of any single software tool.

The first CASE system executes on the Hewlett Packard HP3000 mini-computer using the award winning, commercially available HP/Image database and HP/Vplus forms system. Eight real applications (including itself) have been developed in this way. Four are still in commercial use. This system has a full code generation system, automatically creating error-free Pascal code from the repository specification. No 'experience' tracking facilities are included. For distinction, the CASE tool is called the 'Foreman Development System' (FDS).[1]

The second CASE system, which started development in 1990, executes on high performance PC Networks using an open database system and internal forms system. Over fifty real applications (including itself) have been developed in this way. Commercial installation of the applications number about sixty sites. Each site is configured with between one and twelve of the applications. Development of the applications are still on-going (as requirements are still changing). This system has a more limited code generation system but full 'experience' tracking facilities. For distinction, this CASE tool is known as the 'Elite Development System' (EDS).[1]

[1] FDS and EDS are the commercial property of Thom Micro Systems Ltd.

The Role of the Author

Phase is an idea conceived by the author of this thesis. The author also designed the CASE tools FDS and EDS and was personally responsible for their development. Implementation was performed by a small team of developers working directly for the author.

1.3.2 Recording Design Experience

Recording design experience is a significant component of the Phase system. Although it is very difficult to compute an individual contribution of any element of a development process to the overall result of a software process, intuitively it is felt that the recording of design experience contributes to about a fifth of the overall benefits.

Recording design experience is a concept whereby decisions which are made during the development process, and the rationale supporting the decisions, can be recorded in such a manner that they can be 're-run' at a later stage. The benefits of being able to do this effectively are enormous.

Let us assume that an application is developed (using any software process) by an experienced software engineer. An experienced person will make decisions on certain attributes based upon, perhaps, years of encountering similar situations. Experience, by its definition, can only be achieved by relating to similar situations and by only two methods:

- Relating to previous situations encountered personally;
- Relating to previous situations encountered by other people

Gathering experience personally can be a slow and painful process. The common phrase 'learn from your mistakes' attributes perhaps a greater learning from bad experience than from good experience; however, the consequences of bad experiences may be extreme.

Experience is perhaps best learned from other people. It is passed on by speech, reading and watching. These forms of communication can be extremely slow. An ideal situation, in general terms, would be to 'wire in' the thought processes and experiences of one individual to another, thus allowing experience to be transferred directly. This may be possible in the future, but it is still science fiction in today's technology.

Experiments in this field have been conducted along two major fronts. The first is in the recording of design rationale as decisions are taken; the second is in the technology of artificial intelligence where machines are being 'trained' to become expert systems. References to these techniques are made later in this chapter.

The Phase system takes the approach that 'experience' is held within the development process itself. This 'experience' is gathered by recording the action taken by software engineers, and the reasons for the decisions, as they define and refine a specification. This provides an automated technique which goes beyond simply knowing the final definition of some specification element but also the reasons why an element has its defined properties.

In later activities, as requirements change and specification elements are re-evaluated, the experience 'recorded' by the process about the element can be 'played back' to a software engineer (who may, or may not be the engineer who had been involved previously) who is considering change.

It will be shown later that this information significantly improves the ability to incorporate change into specifications and resultant software.

1.3.3 Monitoring and Validating Change to Software

A third contribution of this thesis, which can be considered as a by-product of the data collection exercise of capturing 'experience', is a method of monitoring and checking changes made to software. Although this technique has been applied to a Phase development strategy, it is generic in its use and can be applied to any software process.

Most literature about software quality suggests that well-trained, highly skilled and qualified staff provide the largest contribution to the quality of resultant software. These personnel are costly compared to lower skilled and less qualified or experienced staff. The overall personnel cost of development equals the average cost of the personnel multiplied by the number of personnel. The economic law of diminishing returns [Smith72] can be used to fix the 'ideal' number of personnel for a given task. Assuming that this number is fixed, the only methods of reducing personnel costs is to lower the average cost of personnel. Whilst it would be regarded as non-viable to reduce the cost of an experienced individual, it is possible to replace

experienced individuals with less qualified or experienced individuals, at a lower cost. This poses the problem of maintaining quality.

The Phase process uses a technique called the Quality Inspection Register (QIR) to provide a cost effective mechanism for checking and validating work carried out by less experienced personnel. This is based upon the judgement of experienced personnel with regard to a complexity of a 'change' task and the perceived ability of a less experienced software engineer. It provides a demonstratable mechanism for monitoring change and maintaining the quality of software.

1.4 Structure of the Thesis

This thesis has ten chapters and three appendices. Chapter 1 is this introduction. We now briefly present the remaining chapters and the appendices.

Chapter 2. *Methods of Working and Related Work.* This chapter describes the background for the methods used to collect and analyse data. It justifies the use of case studies and explains why experiments are inappropriate for this study. Related work is presented. This falls into three main categories : similar paradigms, recording experience and general comments on the design process.

Chapter 3. *Software Requirements and Change.* This chapter expands the notion of the software process in order to identify the role of software requirements within the process. Requirements are classified to provide a definitive understanding of different ways in which requirements affect a specification. This leads to a conclusion why traditional forms of specification can be unsuitable for certain classes of applications.

The notion of 'change' with respect to requirements is regarded as an essential issue in software development. This chapter identifies why changes occur (using 'real' examples) and the different timings in the software process where they are introduced. It is shown that with existing software technology many of these changes do not pose any great problem. There is a

significant problem, however, when changes occur after a software product becomes 'mature'. This sets the focus for the Phase process which is described in later chapters.

A definition of software quality is introduced and an examination is made on ways that changing requirements affect software quality. The effect on cost of change is also considered.

Chapter 4. *Changing Requirements : Two Case Studies.* This chapter focuses on change of software and considers options which may help eliminate or at least reduce its effect on the long term development of software. This is done in relation to two case studies, one relating to technological change and one relating to user requirements change. A simple model of the software process is presented in the conclusion of this chapter.

Chapter 5. *The Phase Paradigm.* This chapter describes the Phase Paradigm. The Phase Paradigm consists of a repository structure which is maintained via a CASE tool and a set of activities which complete the definition of the Phase software process. The repository structure and the relationship between its components are considered in context of the software development activity. For comprehensibility, the development process is considered as a series of 'states', each state representing the current point in the development of an application. At each state there will be a specification and optionally a software product. The Phase process activities are described in Appendix B. Examples of Phase software (the software which is developed using this technique) are introduced.

Chapter 6. *Defining and Reusing Phase Experience.* This chapter presents how the design decisions which are made during the Phase process are captured. The capture of design decisions is based upon the recording of changes to the Phase specification automatically as they occur. The basic data captured includes when, by whom and how often changes were made. The usefulness (and reuse) of the information as 'experience' is increased by an order of magnitude when the recording of changes includes why they were made.

This chapter includes a discussion on the practicalities of collecting the information. The data collection technique was refined four times over a period of five years. During this time,

with each refinement, both the accuracy of the data collected and the information content of the data were improved.

This chapter concludes with examples of data collected and presents an analysis of how the information contributes to the overall goal of the Phase system facilitating ease of change. It is this information which facilitates experience gained during the development process to be 're-run' in the mind of software engineers in a similar manner to 'plugging in' the experience of one engineer to another.

Chapter 7. *The Phase Resistance to Change.* This chapter asks a number of questions about the Phase system in relation to its resistance to change. Many of the questions are answered using actual experiments performed over the past few years. The experiments involve change of technology and change of user requirements. The success of 'transferring experience' is also considered.

Chapter 8. *A Critical Appraisal of the Phase system.* This chapter provides a critical appraisal of the Phase system. The Phase system can be considered as :

- A requirements analysis tool
- A specification representation system
- A software designers productivity tool
- A software project management system

It will be shown how the Phase system scores against goals defined for each of these 'tools'. The definition of the goals is taken from literature. Finally a number of disadvantages to the Phase system are given.

Chapter 9. *Summary.* This chapter presents a summary of the preceding chapters which sets the scene and limitations for the conclusions.

Chapter 10. *Conclusions.* This chapter states the conclusions of this work and identifies possibilities for further development. Finally it will be shown how it is possible to change the

attitude of software developers to accept change by following a software process which does not aim to complete a software product, but aims at the continual satisfaction of 'user needs' in this ever-changing engineering technology.

Appendix A. *The Phase Repository Structure.* This Appendix includes a detailed description of the Phase repository structure which is included for completeness.

Appendix B. *The Phase Development Process Strategy.* This Appendix is provided for completeness. It details the set of activities which form the process model for software development using the Phase Paradigm. This includes the identifiable milestones, working practices and set of heuristics.

Appendix C. *Acronyms.* This appendix lists the abbreviations used in this thesis. Where possible the use of acronyms has been kept to a minimum for clarity.

1.5 The Topics of the Thesis

The thesis discusses three main topics; the nature of requirements and change; the Phase system and process; and reuse of design 'experience'. To learn about the Phase development process, it is only necessary to read chapter 5 and Appendix B. The discussion of the technique and results of reusing experience are contained completely in chapter 6. To understand the philosophy behind the Phase system and its strengths and weaknesses, chapters 3,4,7 and 8 should be included.

Chapter 2

Methods of Working and Related Work

2.1 Introduction

This chapter provides a brief justification for the method used to investigate the impact of changing requirements on software development and the attitude of software engineers. It will discuss the two main methods of obtaining data for research : experimentation and case studies. It will conclude that the most appropriate technique is case study.

A selection of related work is presented in this chapter. After a brief overview, the related work is discussed in a manner structured according to the main topics of the thesis :

- Changing requirements
- Repository based specification systems
- Program structures
- General studies on design criteria
- Obtaining & encapsulating design experience
- Inspection techniques

The chapter concludes with a short summary.

2.2 Methods of Working

According to Pfleeger [Pfleeger94] there are two primary methods of collecting information for the purpose of evaluating new ideas : experimentation and case studies. The significant factor in determining which method is more appropriate is the available 'level of control'. If it were possible to say, produce two functionally similar applications, one using the Phase paradigm and one using a more conventional technique, using application designers with comparable ability, then the level of control would be high and an experimentation technique would be appropriate. This would allow a direct comparison between the results of two 'experiments' in a controlled manner.

Phase was developed within a commercial environment where it was not cost effective to develop software purely for research. Although the development activity was guided by the author, each application developed had to be commercially acceptable. This has resulted in the chosen technique for capturing data relating to development being via case study. The information is not any less valuable, however it must be acknowledged that any conclusions made, must be placed within the context of the environment appropriate to the case study software development company. This can be summarised as a company with between 10 and 15 full time development staff, each with software development experience ranging from between 2 - 10 years. Some senior members are graduates, some junior members have no formal academic training in software development.

2.3 Related Work

The "impact of changing requirements" is considered a very important issue and one which is attracting attention here in the mid 90's. In 1993 it was the main topic of an International Conference [RE93]. At this conference, strong arguments were proposed [Harker93 et al] that it was far too simplistic to assume that requirements could be captured at the beginning of a project. They argue that requirements can only be defined through a process of examination and interpretation, and emerging or changing requirements will be an outcome of greater

understanding of the problem. This simply highlights the case presented by Brooks as far back as 1975 [Brooks75] and again in 1987 [Brooks87].

2.3.1 Changing Requirements

DTI / Proteus

One of the consequences of this conference however, was the instigation of a DTI funded investigation into the impact of changing requirements [Proteus93]. The Proteus project included a series of case studies set up in order to analyse how organisations view the requirements change problem, and to see what organisational structures, procedures and software tools they use to cope with requirements change in ongoing projects. The findings of this investigation suggest that current technology tools tend to be more concerned with the 'cost of change' as opposed to 'management of change' or 'damage limitation' tools which are the 'real need'.

A very recent report [Chudge96] presents a model of the problem of changing requirements in terms of responsibilities and communication between supplier (the developer) and the customer (the user) using some of the interim results of the Proteus project discussed above. The suggestion is made that the basic relationship between 'partners' in a software development project is one of distrust, especially when it involves the 'costing' of changes in what was originally a fixed price contract. Part of this distrust is a consequence of the 'fine line' between what can be considered as further refinements of original ambiguous requirements and actual changes, especially in the latter stages of a commercial software development project.

Parnas

Parnas [Parnas79] is very concerned with changeability of software. He describes all changes as extensions and contractions and proposes a structure of software based upon minimalist subsets. This work tackles the problems of 'change' in a different manner to the Phase system as it is still concerned with traditional methods of specification and structures of programs. The use of minimal subsets is an extension to the concepts of structured programming.

2.3.2 Program Structures

Information Hiding / Object Oriented Systems

It is widely recognised that 'modern' program structures have had a significant contribution to the impact on the quality of software during program maintenance. In particular the concepts of 'information hiding' [Parnas72b] and Object Oriented Systems [Booch91] have provided perhaps the most significant improvements in recent years.

Although both of these topics make significant contributions, this thesis does not discuss either of them in any detail. This is due to the vast quantity of discussions available in other literature.

2.3.3 Repository based specification systems

Blum / Tedium

Blum [Blum91] [Blum93] presents a paradigm for representing requirements in a non traditional manner. He justifies the 'as built' specification approach as being appropriate for systems with open requirements and provides this form of specification in the Tedium system. The Tedium development tool has a similar conceptual structure to the Phase system which is summarised in Figure 2.1. This diagram is reproduced from [Blum93].

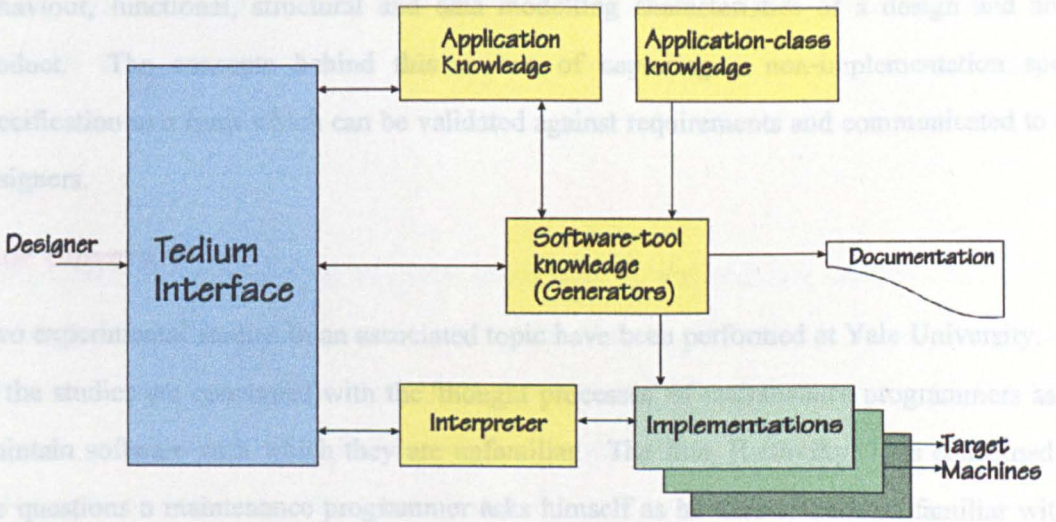

figure 2.1 : The Tedium System

The Tedium system uses an Integrated Engineering Environment to allow a designer to record 'application knowledge' into an 'application database'. This information is used by a number of generators to generate both documentation and implementations on different target machines. The Phase system differs from the Tedium system in the form of the application database. The Phase system contains more specific types of entities (as detailed later) than the Tedium system which is based around a system more akin to a higher level procedural specification language.

2.3.4 General Studies on design criteria

Reeves / Goose

The underlying philosophy of the Phase process which is presented in chapter 4 is that the development follows a pattern consisting of a series of 'states'. The process of development is a process of refining and modifying these states. Each state is broken down into a series of entities or components. Conceptually, thinking about the requirements and design of a state being the collection of requirements and designs of the components making up the state, is similar to an idea discussed in the GOOSE system by Reeves *et al* [Reeves95]. This proposes a design framework which is native to system designers. The state of a system is denoted by a D-Matrix which includes specific (although not process specific) entities which reflect the behaviour, functional, structural and data modelling characteristics of a design and an end product. The concepts behind this is one of capturing a non-implementation specific specification in a form which can be validated against requirements and communicated to other designers.

Yale University

Two experimental studies in an associated topic have been performed at Yale University. Both of the studies are concerned with the 'thought processes' of maintenance programmers as they maintain software with which they are unfamiliar. The first, [Letovsky87] is concerned with the questions a maintenance programmer asks himself as he tries to become familiar with the program code and concludes that there are regular patterns to the way in which a maintenance

programmer will learn about the design. Once this standard pattern of 'self learning' has been determined, it can be used as a template for documentation.

The second study [Littman89] *et al* is concerned with the mental model that a maintenance programmer creates when preparing to perform maintenance on a program with which he is unfamiliar. These mental models relate to the structure of the program and the style of the programmer who wrote it. The conclusion of these experiments is that a maintenance programmers who takes the time to create full mental models will perform maintenance that is less likely to interfere with the quality of the software, than a programmer who only creates a mental model on an ad-needed basis.

2.3.5 Obtaining and encapsulating design experience

Potts & Bruns

The Potts and Bruns [Potts88] [Potts89] method of capturing and reusing design decisions is relevant to the discussions in chapter 3. This work captures design deliberation and considers a design history as a network consisting of artefacts and deliberation nodes. Artefacts represent specifications or design documents; deliberations represent issues, alternatives or justifications arising from these artefacts. A fundamental problem with this work is the practicality of collecting this information and the analysis of the information as 'experience'. This work is subsequently expanded by Lee [Lee91] where explicit goals are included in the representation. Although this does not attempt to solve the data collection exercise, it adds significant improvements to the analysis.

Design Patterns

Design patterns is a concept recently introduced to the software industry by Alexander [Alexander92]. Design patterns is concerned with identifying and documenting features common to any sort of design in a manner that they can be reused as building blocks. Whilst this approach is commonplace in other engineering disciplines, this is the first time that tangible 'building blocks' for design have been documented. This work has been further enhanced by

[Pree94] and [Gamma93] et al. where the building blocks are seen as a method of passing on experience from designer to designer.

2.3.6 Inspection Techniques

Fagan Inspection

The Quality Inspection Register contribution to this thesis for maintaining software quality with less-experienced personnel is similar in principle to the Fagan inspection technique [Fagan77] for checking program source code although in the Phase system it is not source code which is being inspected but changes to specifications.

2.3.7 Summary

The above list of related work is by no means exhaustive however they are major contributors to the topics covered in this thesis.

Chapter 3

Software Requirements and Change

3.1 Introduction

Software requirements and their definition are commonly regarded as the most difficult element in software engineering. This chapter describes the activities of any software process in order to provide a context for requirements definition within the process.

Requirements are not homogeneous and a number of classifications of requirements are discussed. This provides an understanding of different types of requirements and leads to a conclusion why traditional forms of specification can be unsuitable for certain classes of applications.

There are elements of software engineering which are more complex than their counterpart engineering disciplines. These are described to indicate why software engineering is more susceptible to change than other forms of engineering. In software engineering, it is not simply the form of the changes which are important but also the timing in terms of the point in the process when they are introduced. Technology exists to deal with certain types of change at certain points in the process. These will be presented. The major problem with changing requirements occurs when a software product is considered mature.

A definition of software quality is introduced and an examination is made on ways that changing requirements affect software quality. The cost of change is also considered.

3.2 The Software Process

The software process has been introduced as a series of activities which transform a 'concept' or 'need' into a software product and through to product retirement. This process is often referred to the Software Life Cycle. A simple definition is given below. This definition is by no means absolute and the boundaries between the activities are not always clear. The purpose of this description is simply to place the requirements definition into context within the whole development process. The standard process consists of five activities : analysis, design, implementation, testing and maintenance. [IEEE91]

- *Analysis* is the study of a problem (or concept), prior to taking some action. During this activity the properties which the software has to possess are established. This activity defines what the software must do. The result of this activity is the requirements specification.
- *Design* is concerned with how the system is going to accomplish what was defined during the analysis activity. This is a two stage process. The first is where the overall architecture is developed as a high level model of the solution. The second concentrates on determining the data structures and functions and how they are going to be implemented. The design activity uses the requirements specification determined from the analysis activity as the starting point and as a result produces a design specification. The differentiation between analysis and design is not always clear. Some software processes (including the Phase process) combine analysis and design activities into one.
- *Implementation* is the activity which transforms the results of the design phase into instructions for the computer by using a 'programming language'. In the Phase system (and some others) this activity is partially automated by the use of computer technology.
- *Testing* 'demonstrates' that the programs written in the implementation phase satisfy the requirements specification. After successful testing the program is delivered to the users and 'commissioned'.
- *Maintenance* represents an activity which continues from the point of delivery until the point of retirement of the product, making changes to the product as a result of incorrect

implementation or changes to the requirements specification. As it will be shown below, during this time, requirements will change and it is these changes which pose the greatest problem for software developers. In this text, software which is in the maintenance activity will be called 'mature'.

There are several methods to tackle each activity and move between activities. Each documented set of methods forms a software process. Some processes are relatively simple, other significantly complex.

The first and simplest of these models is known as the sequential waterfall process. This was first introduced in [Benington56] and presents the activities as discrete and followed sequentially. A revised process which incorporates feedback and allows iteration to previous activities is more commonly considered as the first real software process. This is described as the waterfall model in [Royce70]. By showing that specifications and implementations are inevitably intertwined, Swartout and Balzer showed how this model was too simple for 'real' development [Swartout82].

Many refinements have been made to this process, the most popular being the spiral model [Boehm86] which has the same basic activities as the waterfall model but permits continual interleaving of the activities as identified by Swartout and Balzer. Here if the implementation activity requires alterations to the specification, the design activity is re-opened, the design modified and the changes propagated throughout the activities as appropriate. The Phase process refines the above approaches even further by cyclically iterating the activities.

3.3 Software Requirements Classification

Software requirements are not homogeneous and may be categorised in many ways. This section presents four classifications of requirements. These classifications are not mutually exclusive, some real requirements can be considered under more than one classification. In addition, it is recognised that other classifications of requirements may be equally valid. These classifications are :

- The Source of Requirements

- The Properties of Requirements
- The Importance of Requirements
- The Character of Requirements

For the purpose of this text, the set of all requirements is known as the *Global Requirements* for software.

3.3.1 The Source of Requirements

Software requirements have two sources. One source is the user (or groups of users), the other source is the technology on which software will be implemented.

User requirements can be considered as a 'wish list' relating to desired properties for the software to achieve the need. The remaining three classifications of requirements are all sub-classifications of user requirements. In this text, user requirements will be called 'requirements for the software'. The term 'user' in this instance does not necessarily indicate a single user but a 'class' of users. This class of users may include users who will eventually use the software ('end-users'); users who may simply be domain experts; or any person with an input into the requirements which can also include the developer.

Technological requirements are imposed by the environment surrounding either the execution of the resultant software or the environment of the development process. For example, the execution of the software is constrained by Operating System (OS) limitations e.g. memory, resource availability; or peripheral specifications e.g. screen size or colour availability, printer feature constraints. The development process requirements may state the need for recalculation or data repair routines. In this text, technological requirements will be called 'requirements of the software'.

3.3.2 The Properties of Requirements

The most common division of user requirements is with respect to the system properties they specify. They are 'functional' and 'non-functional'.

Functional requirements establish the behaviour of the system. They establish the objectives that the product is to meet or the functions that the product has to provide. Generally

functional requirements can be considered in a logical context and can be specified formally. In the Phase system, functional requirements are defined in terms of 'entities' and the 'processes' affecting the entities. In this sense a comparison can be made to the Object Oriented (OO) terminology where entities are OO objects and processes are OO methods.

Non-functional requirements define the conditions that the product must satisfy that are not concerned with its behaviour. For example, the response time from user input to corresponding output; the colour of menus; the standardisation of report headings. These requirements cannot be considered in a logical context. In the Phase system many of these requirements e.g. colours, structures (menu and report), have been recognised with all options to these requirements available as preferences.

Functional and Non-functional requirements may be constrained by limits imposed by external factors. For example, the functional requirement to calculate maternity pay in a payroll system is constrained by the fact that it can only apply to a female employee (by current UK regulations). The non-functional requirement relating to speed of execution of a system will be constrained by the limit of the clock cycle of the hardware upon which it is executed.

3.3.3 The Importance of Requirements

This classification organises requirements according to their relative importance. Three levels are defined. These are Essential, Derived and Implicit.

Essential requirements specify all the properties of the software that must be included for the product to be acceptable. According to [Lehman80], essential requirements are never complete as completeness would over specify and consequently constrain the freedom of design. In the Phase system, essential requirements are all specified in relation to the user interface.

Derived requirements specify features derived from the essential requirements. Derived requirements are never explicitly included in a requirements specification, including them would make them essential. In the Phase system, data table specifications are derived from the user interface specifications.

Implicit requirements are assumed to be a by-product of 'sound engineering practice'. There are always many requirements in this category, only those that demand particular attention are

mentioned. In these instances they are promoted to essential. Implicit requirements often pose a particular problem: as they are never specified it is important that all concerned (the user and the developer) have a similar understanding of implicit requirements. In practice this is constrained by the different levels of knowledge about the application domain and the capabilities of the software in the appropriate technology. In the Phase system, by the nature of the maturity of the process, the capabilities of software can be demonstrated in advance. This provides some degree of coherence in the understanding of implicit requirements.

In some way, implicit requirements exist before any project is initiated. Essential requirements are the components of a specification which are found in 'traditional' specifications. Derived elements are formulated during the design activity.

3.3.4 The Character of Requirements

The final classification scheme qualifies the character of requirements. Two of these definitions have been described in the introduction to clarify the bounds of the application class for which the Phase process has been developed. These are Closed, Open and Abstract [Blum93].

Closed requirements are well defined and stable. They can be completely determined before fabrication commences. In many engineering disciplines there exists a notation in which to express these requirement e.g. a mathematical notation can be used in mechanical engineering applications. Due to the fact that these requirements can be specified precisely, the greatest uncertainty in the development process is the ability of the final product to meet these requirements. The Phase system can be used to develop applications with closed requirements, however the potential of the Phase system is not realised in this instance.

Open requirements are poorly understood and dynamic. They cannot be determined before fabrication (of some form) commences primarily due to the immaturity of the current level of computerisation in the product domain in which the product is being developed. Due to the fact that these requirements are uncertain and dynamic, the greatest uncertainty in the development process is the ability of the final product to achieve the satisfaction of the 'real' needs of the software. The potential of the Phase system is exploited when developing products with open requirements.

Abstract requirements are concepts which have no concrete realisation, for example 'safety' or 'security'. These concepts may be both functional and non-functional and a representation is required to allow analysts to reason about them. The greatest uncertainties in the development process are how effectively the representation scheme captures the concept and how thoroughly the representation is investigated. The Phase system has no facilities for representing abstract requirements.

3.3.5 A Requirements Classification Relationship Summary

Figure 3.1 summarises and demonstrates the relationship between the four classifications of requirements.

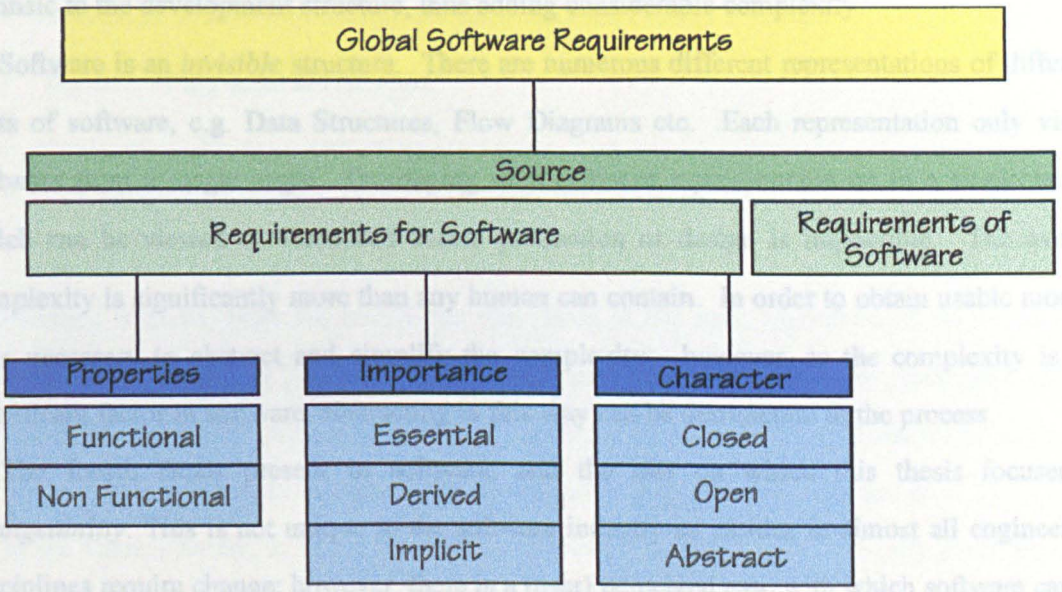

figure 3.1 : Requirements Classification Relationship Summary

3.4 Complexity of Software Requirements

The problems inherent in defining requirements are well documented [Rowen90] [Royce70]. Whilst this can be true of almost any engineering discipline, software is typically regarded as having four 'more difficult than usual' properties. Brooks identifies these as the notions of complexity, conformity, invisibility and changeability.

One of the main reasons for *complexity* in software is the lack of repeating elements. Unlike electronic, civil and mechanical engineering disciples, large projects are not made up by repeating small 'building blocks' a large number of times. Software elements are interrelated in a non-linear manner which means the complexity of a project increases at a much greater rate than that of the physical size of the project.

Conformity of software adds considerable complexity to a system. This relates to the number of interfaces in which a system tends to be involved. The user interface may have to conform to the current 'flavour of the month', ranging from simple scrolling terminals to complex Graphical User Interface (GUI) systems for a similar function. Interfaces to special hardware systems or connected software modules impose rigorous structures which may not be intrinsic to the development structure, thus adding considerable complexity.

Software is an *invisible* structure. There are numerous different representations of different parts of software, e.g. Data Structures, Flow Diagrams etc. Each representation only views software from a single angle. Overlaying each different representation on to a single model which can be viewed or visualised before production or design is impossible. The overall complexity is significantly more than any human can contain. In order to obtain usable models it is necessary to abstract and simplify the complexity; however, as the complexity is the significant factor in software, abstracting in this way can be detrimental to the process.

The fourth factor present in software, and the one on which this thesis focuses is *changeability*. This is not unique to the software industry as entities in almost all engineering disciplines require change; however, there is a (user) perceived ease with which software can be changed which encourages both requests for change and the unwillingness to recognise a large cost associated with a change. It is readily accepted however that changing physical structures, houses, circuit boards etc. will require extensive replanning or retooling and consequently encounter the much higher cost. This is primarily due to the invisible nature of software.

With all this complexity, is it ever possible to produce a requirements specification which contains all the 'user needs'? Brooks [Brooks75] states that it is really impossible for a client, even working closely with a software engineer, to specify completely, precisely and correctly

the exact requirements of a modern software product before trying some versions of the product.

3.5 Why do Requirements Change?

Software requirements will change with time. There are five major reasons identified why requirements for software change. These are :

- If the real requirements are not satisfied ;
- If the real requirements are satisfied;
- A software system will change the environment in which it is used;
- The user of the software changes;
- Computer technology will change.

Real Requirements Not Satisfied

If delivered software does not satisfy the real needs of the user, regardless of the reason, the requirement for change is obvious. In traditional software engineering, the problems could be attributed to any of the process activities; the analysis could have been inadequate, the design failing to meet the requirements, or the implementation failing to meet the design.

In the Phase process, as implementation is a proven computer task, the only place for error is during the combined analysis/design activity. That has happened during the development of Phase applications. One example that is prominent, was the development of a retailing system for coal merchants. In this instance both the user and the developer were 'higher management' who, although they had been involved in their respective businesses for over fifteen years, were far removed from the actual day-to-day tasks in the application domain.

Real Requirements Satisfied

The user requirements will change once a system has been 'used'. If a software product is found to be successful, people try it for new cases at the edge of, or beyond the original domain

[Brooks87]. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it.

A practical example of this recently occurred in an application for an 'Oil Industry' service organisation who had a need for a 'fabrication-shop job costing' system to monitor costs and charges for work. As soon as it was commissioned and the basic job information entered, the potential of the system for shop-floor scheduling became apparent. The data already entered for jobs included relevant fabrication start and finish dates and a breakdown of resource allocation for costing. This information was so relevant to a planning system that the users tried to use the information for the purpose of scheduling and work planning. At this task, the software was poor (it had never been designed for this purpose) and the general satisfaction of the user was diminished. Changes were made to the software and as a result, both planning and costing functions are equally accepted.

Software Changes Environments

Programs interact with their environment and change the original environment by their operation. This has the consequence that user's expectation of satisfactory performance changes as he is exposed to and uses the software system [Giddings84]. Even before delivery, there is a passage of time between requirements generation and system delivery. As users gain more insight into the planned environment their goals and expectations change.

Recently an application was being commissioned for a bakery company with many shop retail outlets. The current 'real problem' was the quick and accurate analysis of daily sales data in order to make better management decisions regarding manufacturing quantities for products with such a short shelf life. Collecting the daily sales figures was not a problem as they were submitted on returns to head office at the start of every day. The application commissioned matched the requirements in every way. Shortly after commissioning however, the user became dissatisfied : although the analysis of the sales data was reduced from hours to seconds, the data entry time of the sales returns had not changed (as there was no requirement for change at the time of the analysis). It took one hour every day to enter the returns for all twenty sales outlets. Using the 'old' system, this was only 20% of the total sales analysis time, with the new software

it was now 99% of the time. A requirement for change for automatic data entry from the sales tills via modem resulted as a consequence of installing the software.

The User of the Software Changes

All software, like everything else, is subject to the human characteristic of individual taste. This is excentuated if the software is particularly 'human oriented' (like IBIS Software). Even requirements which are defined by legislation still have elements open to interpretation, for example, screen layouts and report styles. These cosmetic entities are subject to individual appreciation and, like an opera, does not have universal appeal.

Changing users therefore has a significant impact on changing requirements. There are three major instances where this is highlighted.

The first is where software is aimed at the mass-market as a 'standard package'. In this instance there is an unknown quantity of users with unknown tastes and preferences. Software aimed at this level (or ending up at this level due to popularity) will have to be suitable for the general case in terms of behaviour and 'middle of the road' in terms of non-functional design.

Designing software like this inevitably leads to a 'Jack of all Trades, and Master of None' syndrome leaving perhaps no user completely satisfied without continuous (although usually minor) modification for increased flexibility. An example of this was the Elite Payroll software module which has an installed base of 31 sites. Although the core requirements of payroll are defined by legislation, 29 of the sites required at least 1 modification to enhance user satisfaction. The majority of these changes were cosmetic, usually to reports. [It could be argued that these changes were not absolutely necessary however for commercial sense, incorporating these changes gave the client a greater feeling of 'Value for Money' at the prices paid.]

The second main reason for change of users is when the 'user organisation' has a culture of change. A major example of this is government institutions where users are elected or reallocated on a regular cycle. In these instances the users will change (as will the legislation based upon the different government manifestos) every few years. This was highlighted in the Elite 'Homeless Persons' software module used by local councils to maintain registers and manage the waiting lists and housing allocations for homeless people. The software had

maintenance performed in the period April-May every two years - the same period where departments redeployed personnel as part of a continuous staff training and 'reshuffle' program.

Even if a company does not have a culture for changing personnel on a regular basis there will always be changes relating to either promotions or employees changing jobs. This is the third major reason for change of users. This was highlighted during one Phase installation where the main user was promoted two weeks before delivery of the software. His replacement as head of the implementation team had very different ideas on the solution. As a result the project was eventually abandoned.

Computer Technology Changes

Requirements of the software change according to the current technology of the hardware platform or operating system. Changes in hardware technology have advanced at the fastest rate of any engineering discipline [Brooks87]. Even if user requirements for software change very little, the machine vehicle for which the software was first written will change, be it new computers, or at least new disks, displays or printers as they come along.

An example of this is the subject of the first case study, presented in chapter 4.

3.6 When do Requirements Change?

The timing of introducing requirements change has a major impact to the complexity and cost of incorporating these changes. The cost will be discussed later in this chapter.

From personal experience, changes introduced to software occur in different ways at three definitive points in the life of the software. These are:

- during the main analysis and design activities;
- during the commissioning activity;
- after delivery and during the maintenance activity.

During Analysis and Design

It has been suggested by Giddings [Giddings84] that at the start of a software development project, the user only has a vague idea of requirements. The requirements are open. At this

stage, requirements are conceptual, lack detail and do not form a precise, well thought-out plan where the implications on the surrounding environment have been properly considered. Requirements at this stage do not so much change as go through a process of refinement.

This is the basis of the Phase process and many examples could be discussed. One project, concerned with software for housing associations has a total timetable covering three years. This project was divided into six smaller applications. The 'users' in this instance were domain experts but had little knowledge regarding the power of computerisation. Analysis and design meetings were significantly longer than with more computer-literate users. At the start of the project there were no written requirements.

After the first analysis meeting, six pages of notes were taken and the application was conceived as 'easy'. At the second analysis meeting, based on a refinement of the first, an additional eighteen pages of notes were taken and the application conceived as 'difficult'. The third had an additional five pages of notes and the fourth an additional two. The overall functionality of the software (its scope) did not increase during this time, the increased specification related only to the level of detail defined.

During Delivery

When a project is being commissioned, actual changes can be more easily identified. Typically they are in the form of additional points raised, based upon data which is at the periphery of the existing requirements. As the requirements and/or design of a system are more fully defined and understood, time and attention are available to consider examples of data which will not exactly fit the system, but are so close that (seemingly) minor changes can allow them to be incorporated.

This is fuelled by three elements :

- As mentioned earlier, due to the invisibility of software, true understanding of a system is only achieved when the system is delivered. At this point the concept has a tangible representation;
- Real data contains a much wider variety of examples than that usually considered in the earlier stages;

- A wider variety of users are exposed to the software, each bringing a different viewpoint and/or perception of how it should be.

To find an example of the first type of element it is necessary to return to development prior to the Phase process. During the late 1980's before the creation of the EDS, development at TMS on micro computers was more akin to the traditional software processes. One small system, for a shipping company breaking into the property market required an application for maintaining information on leased property. A full analysis activity was performed and a detailed requirements specification was prepared. This was accepted by the users who appeared at the time to understand it. When the software was commissioned it was rejected by the users as the 'conceptual picture' of the software in the minds of each user was different from the application produced.

The problem of data has manifested itself a number of times. One example relates to a specialised accounting application, created for a firm of accountants. This was developed using the Phase process and resulted in a 'near perfect' specification. The problem related to an implied requirement, the size of the data field for 'money' type data. The very first 'real' data could not be entered, the assets of the 'client company' was £3,000,000,000.00, the maximum size of the field was 10 characters.

Introducing new users to a system as it is being commissioned poses perhaps the greatest source of requirements for change. This has been discussed in detailed earlier in this chapter. Two additional examples are prominent:

The first relates to a project for a fabrication company who have multiple plants around the world, each plant has an identical manufacturing process. The application software was designed in conjunction with one of the (geographically local) plants with the intention of providing the same software for all the other plants. The software was accepted by the local plant and rejected by all the others.

The second example relates to a firm of electricians with a head office and two subsidiary offices. The application for a purchase ordering system was designed in conjunction with the department at head office with the intention of the software being installed at a subsidiary office with the premise 'that is how it (the way in which head office wanted the buying to be done) has

to be done'. Even with extensive changes during commissioning the application was made redundant after one year.

After Delivery, During Maintenance

Changes to requirements occur, after the software has matured, largely as a victim of its own success. With the completion of a project, consideration is given to the 'next phase'. This may be in the form of postprocessing of data output from the system or preprocessing of data input to the system. Whilst it can be argued that quality software should be regarded as a black box and not affected by changes in the inputs and outputs, it is extremely likely that

- the form of the inputs or outputs will change to cater for different module interfaces
- additional information will be required to be collected from the inputs, to be passed to the outputs simply for the postprocessor

In addition, changes to the user interface, hardware or operating platform may change without a change in functionality at all.

An example of this type of change is the subject of the second case study in chapter 4.

3.7 Requirements & Software Quality

The quality of software has two major definitions. The traditional image of software quality relates to the physical build of the software [Daily92]. In this definition, quality software would have the characteristics of being well structured, properly commented, fully documented etc. The second definition [Floyd83] [BTRL90] [Agostoni88] relates the quality of software to the effectiveness of the software in meeting the users requirements. In this document the quality of software will be related to the ability of the software to maintain a satisfaction of user requirements as changes (both user and technological) occur without having to start afresh with each generation

It has been argued [Floyd83] [BTRL90] that the quality of software can be thought to be deteriorating during maturity. Based upon the concept that software quality is related to the 'closeness' of software to its requirements, the fact that mature software is primarily static and

that requirements are changing, inevitively leads to the gap between software and its requirements becoming wider. The concept of the "Software Death Cycle" [BTRL90] is an interesting study concerned with monitoring this gap and measuring the cost-effectiveness of standard maintenance techniques. It proposes a method for determining when software should be considered as at the end of its useful life.

3.8 The Cost of Changing Requirements

It is common belief that 70% of the total costs of software are incurred after it has been developed. This cost is spent in the correcting of errors and in the enhancement of the software to meet needs which were not identified before delivery, either due to bad analysis or the essential nature of change as previously discussed.

This poses two major problems :

- It has been shown that the cost of software change increases ten-fold with each activity in the software process [Boehm88]. The fact that 70% of change is during maturity results in a real cost being orders of magnitude greater than costs which could be encountered in theoretical development and cost estimating.
- Commercially, although perhaps only 30% of effort is required before delivery, typically 80% to 90% of the software will be charged. This has the effect that when 70% of the effort is being expended on software, there is an income of only 10% to 20%. It would not make commercial sense to 'admit' that this 70% of work will happen after delivery. Whilst this may show huge profits on software sales, the overall margins are significantly lower.

These two factors contribute significantly to the 'bad reputation' generally associated with the software industry as a whole.

3.9 Current Technology

This section summarises the current technology with respect to specifications and methods of dealing with changing requirements. It is presented in general terms, summarising the typical industrial case only and included simply as an overview.

3.9.1 Specification

Typically requirements specifications consist of diagrams (e.g. data flow, entity relationship), formal and logical proofs and subjective statements. Only user requirements are included, technological elements are implied. There is a clear distinction between the functional and non-functional requirements. A traditional specification identifies only the essential requirements and identifies them as closed. Many real applications require complex products which intrinsically include requirements which are open and abstract, these are generally ignored. The choice of which requirements are explicit, derived or implied is subjective, the selection being more akin to perceived current day priority than product-specific needs.

Specifications may be maintained manually or with the help of CASE tools. Even in the latter instance, specifications are 'separate' from executable programs. This leads to 'drifting' between specifications and programs. The more mature a program, the less likely that changes made will be reflected in the specification, primarily due to the cost/benefit ratio of updating specifications, and time pressures to install software. This in turn, leads to the only accurate specification of a software product being contained within the complexity of the program source code only.

This leads to the serious question of the suitability of traditional specifications to meet its objectives within the role of developing quality software.

3.9.2 Dealing with Change

Change During Design

Many existing analysis techniques which are successfully in operation, iterate processes of refinement until a more concrete requirements definition can be formed. This is more concrete

in the minds of both the user and the developer. One of the most effective techniques is based upon forms of software prototyping [Floyd83] which will be discussed in greater detail later in this document.

Personal experience in commercial IBIS software development using software prototyping have resulted in a pattern where a 'usable' requirements definition is generally available after the third iteration. In this sense, 'usable' equates to a cost effective balance between 'gains & effort (cost)' of additional iterations. An intuitive representation of refinement of requirements is given in the following diagram. This shows how each iteration correlates to the closeness of the requirements definition to the actual user needs.

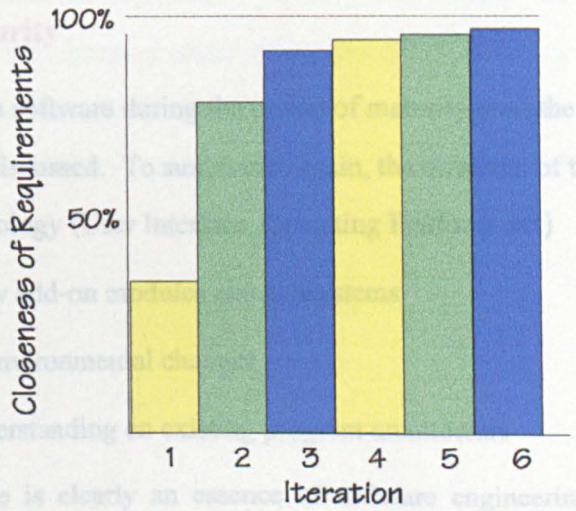

figure 3.2 : Requirements refinement with successive development iterations

At this stage it is not relevant to discuss the form of this prototyping technique or why it is possible to refine requirements in this way in three steps. Suffice to say, methods exist which cope with this type of change.

Change During Delivery

Minimising the effects of change at this stage of the development process is largely down to management techniques and being aware of the problems. The three major elements discussed previously all have fairly straightforward answers (in theory).

Reason for Change	Theoretical Solution
Understanding a system only after having hands on experience	Rapid Prototyping techniques help reduce this factor by allowing users access to a form of the software earlier in the development cycle
Real data having a wider range of values that typical test data	Explicitly analysing real data reduces this problem
Introducing new users viewpoints into the system at a later stage	Involve more users earlier. The use of rapid prototyping helps here.

figure 3.3 : Dealing with change during the implementation phase

Again at this stage it is sufficient to note that the impact of these changes can be minimised.

Change During Maturity

Changes associated with software during the period of maturity pose the greatest problem of the three types of changes discussed. To summarise again, the elements of this problem are :

- Changes in technology (User Interface, Operating Platform etc.)
- Integration to new add-on modules and subsystems
- Changes due to environmental changes
- The effort in understanding an existing program architecture

Change at this stage is clearly an essence of software engineering, the consideration of which seems to have been relatively ignored in the literature. Perhaps this is one of the reasons why progress has been slow.

It seems that change at this stage is left as a function for a maintenance programmer who typically was not a member of the original development team and therefore possibly least qualified or competent to consider all the implications of change; or systems become discarded for new replacements systems which (depending upon any salvageable elements) cause a costly duplication of previous effort without an increase in functionality.

3.10 Conclusions

Whatever the individual details of current development process models, the basic principle exists of a 'requirements' which can be determined and subsequently transformed by some (process dependant) method into a software product.

It is proposed that these software processes are fundamentally wrong for developing quality software as they lead only to 'short term' solutions, principally because of their lack of due consideration to the inevitable continual change of requirements.

Change is an essence of software engineering, the ignorance of which leads to unsatisfied users and exasperated software engineers. The only way to alter this negative attitude is to recognise the importance of changing requirements and develop software using a process which focuses on changing requirements as a central issue.

One such process is the Phase process.

Chapter 4

Changing Requirements : Case Studies

4.1 Introduction

Two case studies are presented as a foundation for the Phase paradigm which is described in the next chapter. These case studies provide objectives for improved software processes and justify the structure of the Phase paradigm. Understanding the culture of 'change' is the first step in controlling it.

In previous chapters, the essence of changing requirements in relation to software development has been introduced. In particular the problem of change of a mature software product is highlighted as a major issue in current software development technology. The reason for change of a software product comes both from the need to keep up with technological 'improvements' and the need for additional functionality.

This chapter is divided into two sections each with a corresponding case study. The purpose of the first case study is to illustrate how, in a commercial environment, computer technology has forced software to be updated over a period of approximately ten years. This case study examines how one commercial suite of software programs developed to incorporate technological changes. This actual development strategy is related to an 'ideal' development strategy and observations made on the differences. An analysis of these observations help formulate a theory of how to migrate software for technological reasons.

The purpose of the second case study is to illustrate how, in a commercial environment, changing user requirements can impact a software product. This demonstrates limitations in existing development practices and provides some basic ideas for an improved software process.

Conclusions from the case studies are used to highlight essential elements of software development which are commonly excluded from current software process models. A simple abstract model of the software process is described which explicitly incorporates these essential elements. This leads to the underlying philosophy of the Phase paradigm.

4.2 Case Study #1 : Change relating to Technological Factors

This case study is concerned with the impact of technological changes on software and is used to illustrate the essential difficulty in keeping software technologically 'up to date'. In order to determine how technology, relating to small IBIS software products, changed over a period of ten years a study was made of the development strategy of a commercial software development company. In this study the reasonable assumption was made that the software products developed at any period in time reflected the requirements of the commercial market.

Figure 4.1 illustrates a 'product history' showing the major versions of an Accounting and Costing package over a ten year period from 1981 to 1991.

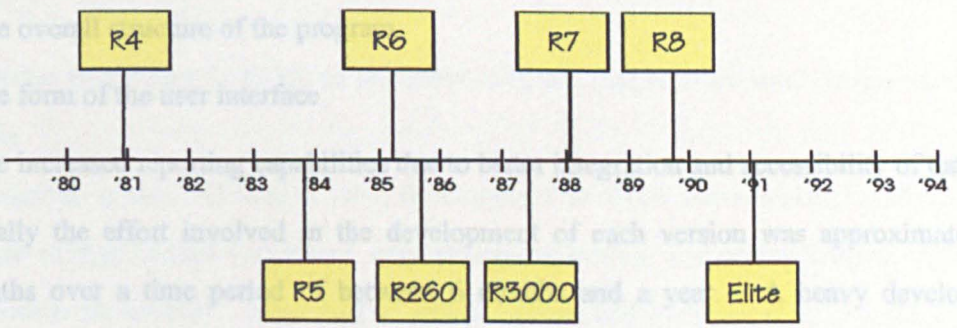

figure 4.1 : History of product development

The primary change factor in this instance is technological change concerned with hardware platform and operating system changes. These are summarised below.

Version	Reason for Change
R4	Original commercial version. Written in Interpreted BASIC. Single User. OS limits program to 32K.
R5	Written in compiled BASIC : never released commercially
R6	Compiled BASIC using ASCII file structures. Multi-user capability.
R260	Mini computer platform using relational database. True multi-user
R3000	Small Mainframe computer using relational database. True multiuser. Power for large number of users and database transactions
R7	Written in PC 4GL using 'simple' database files. Language became obsolete
R8	Re-write of R7 in more powerful PC language with open database structure files
ELITE	Requirement for modular design for larger applications. Colour standard interface.

It is important to note that the overall functionality of the package did not increase significantly during this time. The general requirements of an accounting system are reasonably well defined. What did change as an impact of changing technology, from a user viewpoint was primarily :

- The overall structure of the program
- The form of the user interface
- The increased reporting capabilities due to better integration and accessibility of data

Typically the effort involved in the development of each version was approximately 18 man-months over a time period of between 6 months and a year. A heavy development activity without achieving additional functionality would not have proved cost-effective. A further study into the development of the package was performed to try and establish how each version was produced.

An ideal sequence would be that each subsequent version would be based upon the experience of the previous version where only the 'technological changes' would require attention. An ideal progression would be as displayed in figure 4.2.

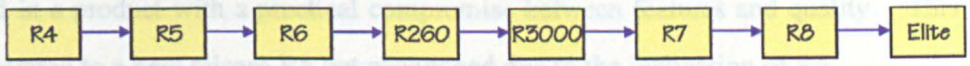

figure 4.2 : An ideal development path

It was found, however, that the relationship between the versions was not sequential but followed the pattern displayed in figure 4.3.

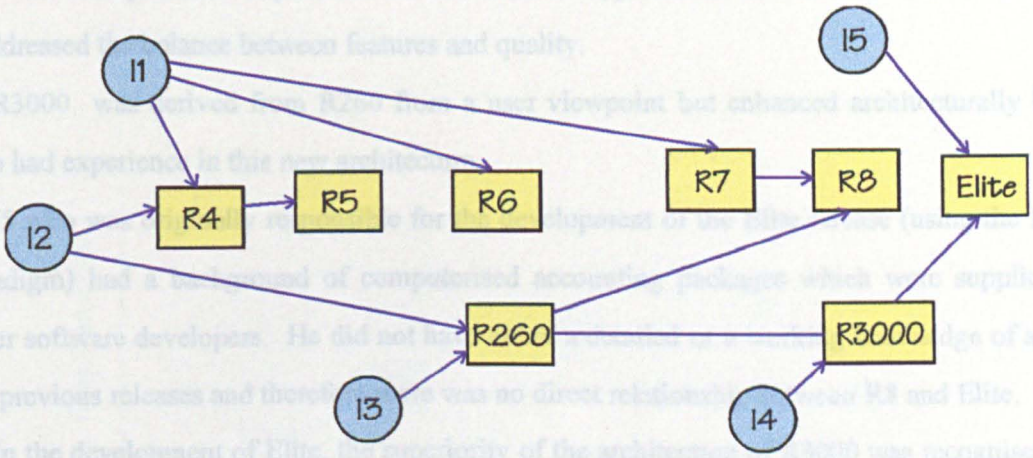

figure 4.3 : Actual development path

The circles (numbered I1 to I5) in the above diagram represent external design input from individuals.

The original system R4 was a joint development between individuals I1 and I2. The architecture of this system was based very closely to manual accounting ledgers replicating existing forms one-on-one. Individual I1 had a background of experience in sales and was 'feature motivated'. He was very close to potential users before the development exercise was started and consequently promised features (without perhaps realising the consequences on structure and implementation of incorporating these features) in order to encourage the potential

user to buy. Individual I2 was from an engineering background and more methodical and systematic in design. He concentrated on detailed design and determined the constraints on the features due to the current software and hardware technologies. In many ways he was responsible for the 'built in' quality of the product. The combined approach of I1 and I2 resulted in a product with a practical compromise between features and quality. This product was migrated to a new release R5 but abandoned due to the instigation of R6.

The development of R6 and subsequently R7 was performed primarily by I1 (feature motivated). The absence of I2 meant that the equilibrium of feature and quality was unbalanced (to the detriment of quality) with the result that R8 became a necessity. This incorporated the architecture of R260 developed concurrently by I2 and I3 enhanced by some of the features of R7. I3 had practical experience in the domain application of accounts and as a result readdressed the balance between features and quality.

R3000 was derived from R260 from a user viewpoint but enhanced architecturally by I4 who had experience in this new architecture.

I5 who was originally responsible for the development of the Elite release (using the Phase paradigm) had a background of computerised accounting packages which were supplied by other software developers. He did not have either a detailed or a working knowledge of any of the previous releases and therefore there was no direct relationship between R8 and Elite. Early on in the development of Elite, the superiority of the architecture of R3000 was recognised and the combined experience of I4 and I5 resulted in the current Elite version.

This shows that a number of versions were based upon the experiences of individuals and not directly upon previous versions. In practice, the development of the other versions were only based on previous versions as they had members on the development team who had been involved with the previous versions.

This example seems typical of development of many versions of programs (possibly even developed concurrently for different hardware platforms), especially for smaller projects. This common problem occurs due to the lack of a suitable 'specification'. There is no concept of 'company experience', simply the experience of many individuals.

4.3 Observations

This case study raises the following question:

> *If we develop a software program from a set of user requirements and a significant change in technology forces a major development exercise, why is it difficult to have a simple upgrade path from the existing system to the new one?*

In order to formalise an answer, let us look firstly at a simple diagram showing the relationship between requirements, programs and the impact of time; and then discuss the factors involved in implementing a program P from a user specification S.

Requirements & Programs

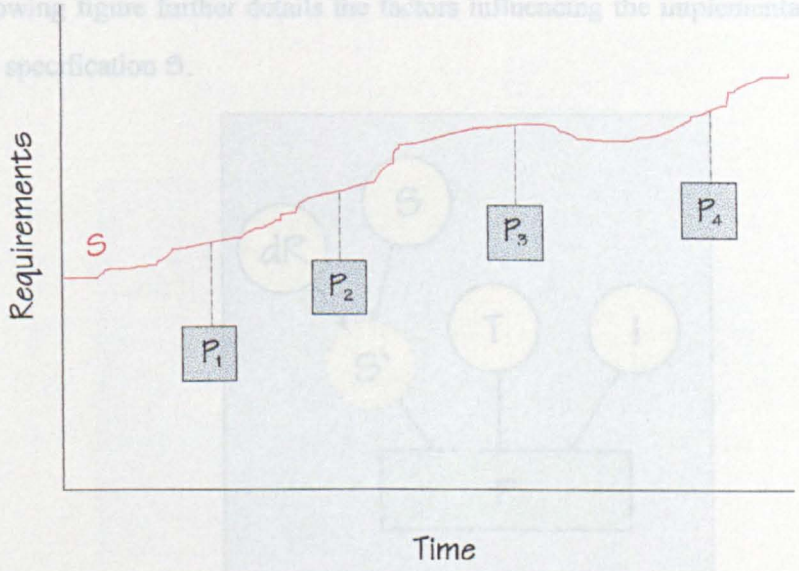

figure 4.4 : Requirements and Programs

Figure 4.4 provides the context for the following discussions on the factors influencing implementation of a program from a specification. In this figure, the changes in the requirements of a system is indicated by the line labelled S. This is purely illustrative, a more

detailed discussion on this line is given at the end of this chapter. In an ideal world, a program P would satisfy the requirements S at all times. Experience however, shows that :

- there is a time delay to implement changes
- programs cannot be continually modified, eventually they will requirement abandonment

Programs can be treated as an instantiation of a set of requirements at a given period of time. Continually satisfying requirements will require a number of different instantiations at different periods of times (indicated by P1 to P4 in figure 4.4).

To answer the question at the start of this section, we need to examine the factors influencing an instantiation, the implementation of a program P from a specification S and the relationship between successive programs.

Implementing P from requirements specification S

The following figure further details the factors influencing the implementation of a program P from the specification S.

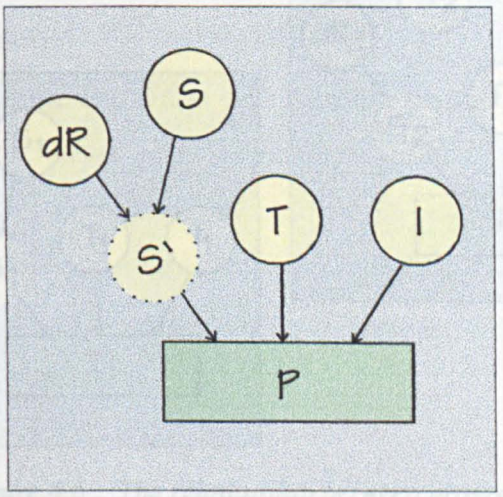

figure 4.5 : Factors influencing implementation of program P from specification S

Figure 4.5 introduces a number of components

- S is a specification, requirements for the software
- dR is a displacement of requirements which transform S into specification S`

- S` is the actual user specification which is represented by program P which incorporates the essential element of change
- T is the technological factors influencing the program requirements of the software
- I is the set of individual experiences of the developers who create the program.
- P is the resultant program

A program P is therefore the result of combining the components T and I with S`. S` is a specification which has been transformed from S by dR. It should be noted that there are other factors involved in software development e.g. the software process itself, however for simplicity in the following text it shall be assumed that its effect is 'constant' and ignored.

The above illustration is expanded in figure 4.6 which shows two product developments P1 and P2. P2 represents a 'future' generation of P1 in a different technology. This figure will be used to help analyse the expected relationships between the components.

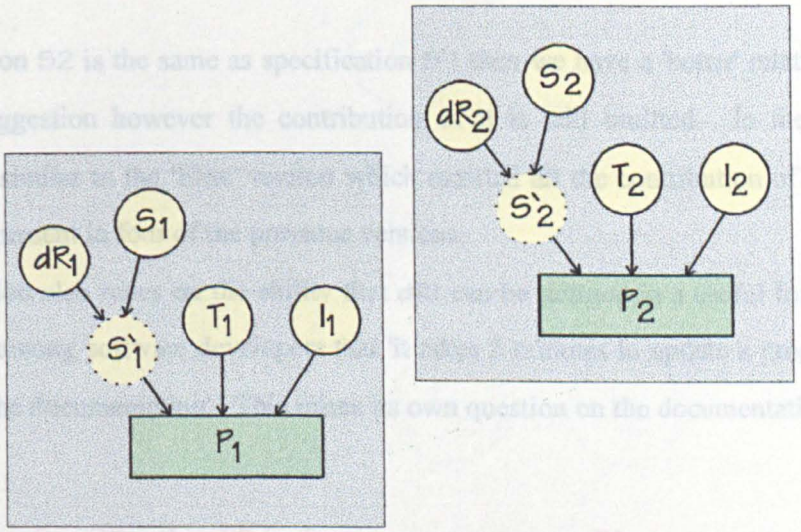

figure 4.6 : The relationship between programs

In figure 4.6 a relationship is shown between P1 and P2 however no relationship is shown between the components of P1 and P2. This is intentional. It would seem obvious that relationships exists, however, the nature of the relationships are not as clear as may first be thought. Possible relationships are examined below. In the following analysis, the assumptions are:

- The specification S1 exists in a complete and usable form
- P2 must be 'derived' in some way from P1 as P2 is a future 'generation' of P1
- The specification S1 is common to P2 as well as P1
- Technology T2 is significantly different from T1
- Individual(s) I2 is/are possibly different from I1
- Requirement changes dR1 and dR2 will exist

Suggestion #1 : Specification S2 is the same as specification S1

If the specification S2 is the same as specification S1 then we do not have an 'ideal' program P2 because dR1 (and thus S`1) and I1 have been ignored. Their contribution to P1 is missing from P2. T1 has also been ignored but as T2 directly replaces T1 this can be regarded as a benefit.

Suggestion #2 : Specification S2 is the same as specification S`1

If the specification S2 is the same as specification S`1 then we have a 'better' relationship than the previous suggestion however the contribution of I1 is still omitted. In the case study example this is similar to the 'Elite' version which omitted all the contribution of I1 (as figure 3.3) which was present in four of the previous versions.

This suggestion also relies on the ability that dR1 can be defined in a useful format. It is a common adage among software developers that 'it takes 5 minutes to update a program, but an hour to update the documentation'. This raises its own question on the documentation format of dR1.

Suggestion #3 : Specification S2 is the same as program P1

In this suggestion it appears that we are incorrectly comparing an equivalence between two different types of entities. For correctness suppose a new entity is created called A, where A can be defined as the complete 'As built' specification of program P. A1 is related to P1, A2 is related to P2 by the same definition. In this way S2 can be the same as A1.

This appears to be an better solution, a perfect solution would be A1 with the components of T1 removed. This theoretically perfect solution is illustrated in figure 4.7.

Achieving such a solution depends principally on the ability to recognise and define T1, I1 and A1 in such a way that they can be manipulated and re-used. Practical suggestions are not obvious!

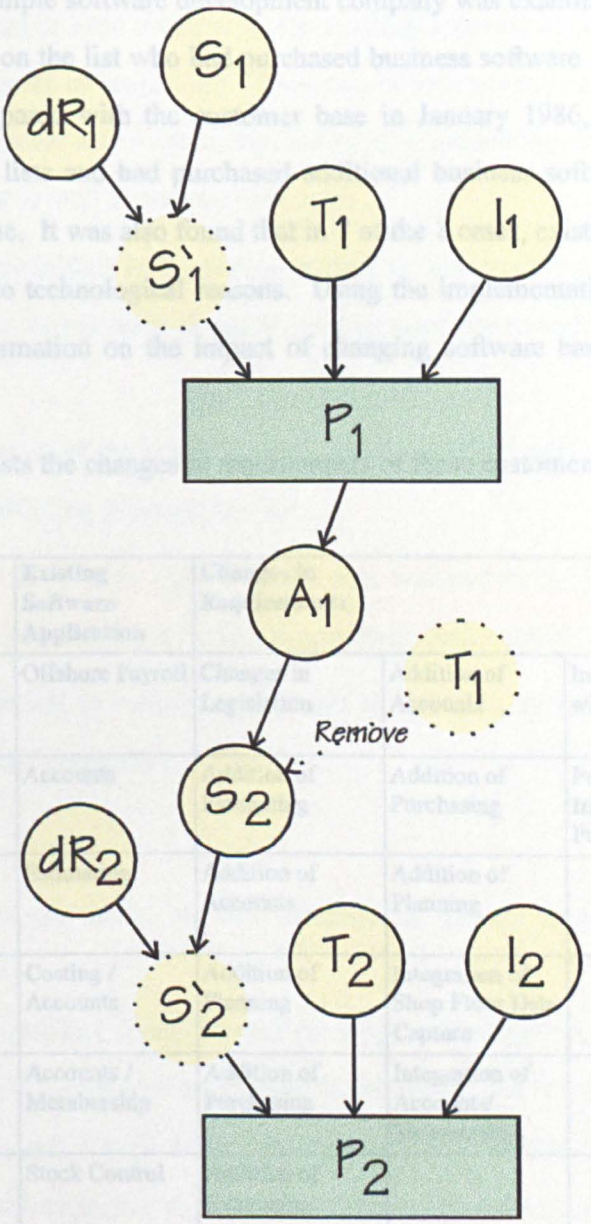

figure 4.7 : The 'perfect' software migration solution

4.4 Case Study #2 : Change relating to change in User Requirements

The second case study was set up to examine the impact of change in user requirements. The customer base of our example software development company was examined. At January 1996 there were 82 customers on the list who had purchased business software in the past two years. When this list was compared with the customer base in January 1986, it was found that 8 customers were on both lists and had purchased additional business software modules during this 10 year period of time. It was also found that in 7 of the 8 cases, existing software modules had been upgraded due to technological reasons. Using the implementation history of these 8 customers provided information on the impact of changing software based on changing user requirements.

The following table lists the changes in requirements of these customers.

Customer Business	**Existing Software Application**	**Changes in Requirements**		
Offshore Oil Personnel Management	Offshore Payroll	Changes in Legislation	Addition of Accounts	Integration of Costing with Accounts/Payroll
Electrical Contractor	Accounts	Addition of Estimating	Addition of Purchasing	Purchase Invoice Integration of Purchasing/Accounts
Sheet Metal Fabricator	Estimating	Addition of Accounts	Addition of Planning	
Offshore Fabrication	Costing / Accounts	Addition of Planning	Integration of Shop Floor Data Capture	
Historic Castle Maintenance	Accounts / Membership	Addition of Purchasing	Integration of Accounts/ Membership	
Timber House Frame Manufacturer	Stock Control	Addition of Estimating		
Cardboard Box Manufacturer	Accounts	Addition of Stock Control	Addition of Order Processing	Addition of Purchasing
Steel Industry Ceramic Manufacturer	Production	Addition of Stock Control		

It can clearly be seen that the majority of changes in the above examples are based upon either:

- Additions of software modules
- Integration of new or existing software modules with existing software modules

In every instance, changes were required to the existing software applications. The degree of change reflected the degree of integration. These can be summarised as :

- Collecting of additional information required by new module eg. Information relating to projects for costing as well as statutory accounts
- Changing the source of data from existing module to new module eg. Purchasing Invoice matching supplying information direct to existing accounts module as opposed to direct input.
- Changing the structure of the information to be accessible by new module eg Estimating information now affecting planning module
- Scrapping 'simple' parts of the system which were replaced by the 'complex' new module eg Simple stock control replaced by full 'accounting' stock control.

Due to the problems inherent in retrofitting changes to software, in each instance the inbuilt quality of the system would deteriorate.

A Detailed Example

The first entry in the above table, for the Offshore oil payroll management company is examined in closer detail.

In this example, the user had a requirement for a specialised payroll system to handle the complexities of offshore oil workers who are paid in an uncommon 4 weekly cycle. In 1985, a computerised system was developed. Legislation changes were enforced by the government in 1991. As technology had improved significantly since 1985 and a multi-user input required, the program was rewritten at this time. In 1992, a management accounts module was introduced (replacing a manual system). Changes to the payroll system were minimal and reflected a change in trial balance structure. In 1995 it was decided that a form of project costing was required which would provide a greater degree of management information. This not only involved further analysis of information output from the payroll system, but required additional

data to be input into the system. Within our example software development company, the same technology was used from 1991 to 1996, therefore we can treat the development in 1991 as the starting point for our analysis of changes affected by user requirements only. As the addition of the accounts module had only minimal effect of the payroll system, these will be ignored from the following example.

In figure 4.8 below, point A is the start of the development process (1991).

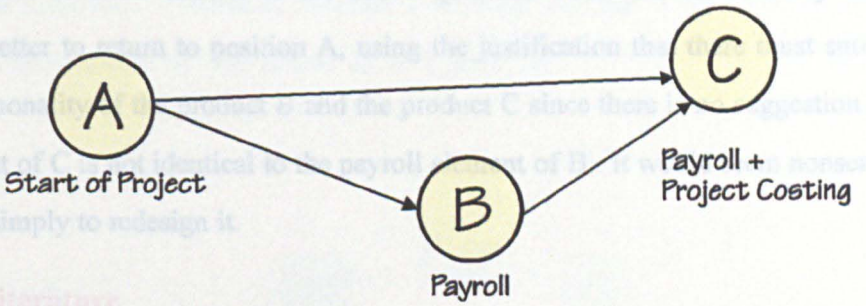

figure 4.8 : The paths for subsequent development

Point B indicates the system developed during 1991. The development strategy followed a path AB where all design decisions were taken to achieve the goal at B. Point C indicates the amended requirement to include project costing in 1995. Starting at point B the development would follow the path BC; however, if C was the original goal the development would follow the path AC.

An Analysis

Intuitively, it is felt that the product C would be in some way 'better' if it had been developed along AC than BC. 'Better' in this sense would mean 'less complex'. Undoubtedly, many of the new elements required in C would have been 'bolted on' to product B as opposed to 'built in' if the product had been developed along AC.

If this is expanding to points D,E and F, all representing additional changes in requirements, it would surely result in a grossly inferior system than that developed in AD,AE and AF respectively.

4.5 Observations

This case study raises the following question :

From the starting point of B, would it be better to return to point A to develop the product C?

The 'immediate reaction' answer to the above question would almost certainly be that it would not be better to return to position A, using the justification that there must surely be a degree of commonality of the product B and the product C since there is no suggestion that the 'payroll' element of C is not identical to the payroll element of B. It would seem nonsensical to throw away B simply to redesign it.

Relating to Literature

Potts and Bruns [Potts88] improved upon a previous idea to record the design process as a series of artefacts, questions, alternative solutions and justifications to answers. The thesis behind this was that a design could be retraced at a later stage, perhaps to find the root cause of failure in some way or to use as a training exercise to educate less experienced software designers, by communicating design decisions made by more experienced designers.

This model was later improved upon by Lee [Lee91] to include explicit goals which could be used as a guide when making the design decisions. The concept behind this was that should the goals be altered, the design could be re-run and each of the decisions re-evaluated in terms of the new goals. A diagram of a design process can be represented as figure 4.9.

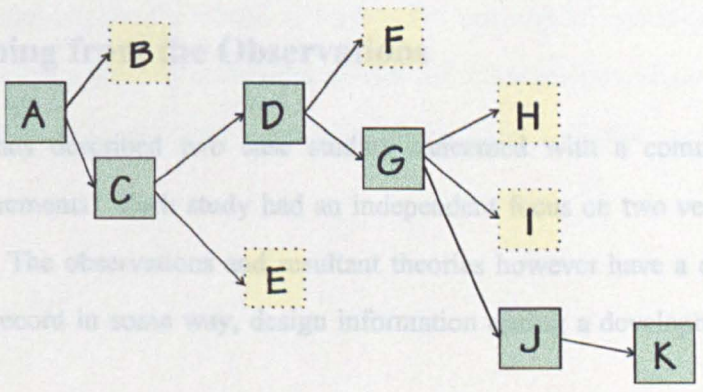

figure 4.9 : A Potts & Bruns design graph

In this diagram, the solid boxes represent stages in the design at which design decisions are made. The lines leaving each box represent the alternative answers to a design question. Dotted boxes indicate answers which were discarded.

In theory, when faced with the situation given in our second case study of changed requirements, it would be possible to return to the original starting point and follow each of the design points re-evaluating each alternative solution. In this instance re-use of a previous design would be achieved. Eventually however a different alternative would be chosen and a new path would then have to be 'trail blazed' as in figure 4.10.

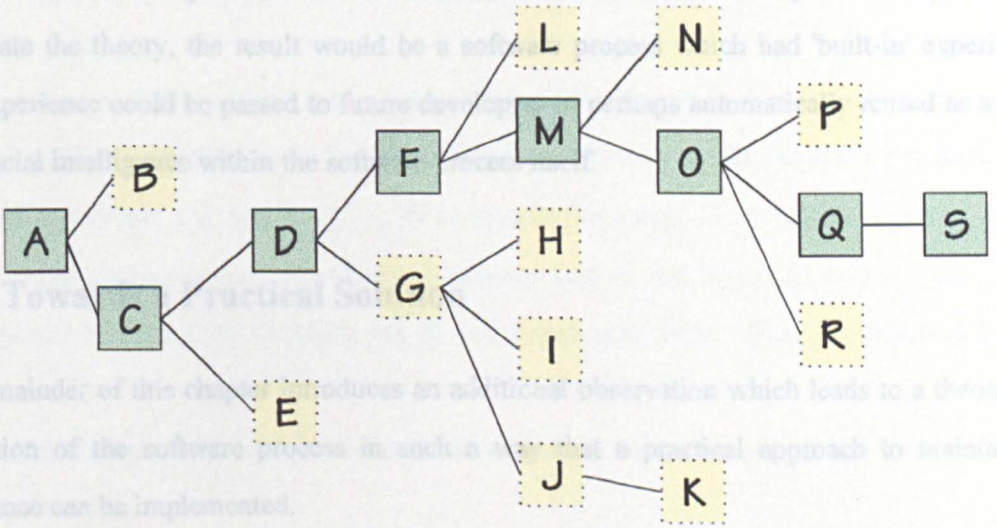

figure 4.10 : Re-evaluating a design graph for different goals

4.6 Learning from the Observations

This chapter has described two case studies concerned with a common theme: changing software requirements. Each study had an independent focus on two very different aspects of requirements. The observations and resultant theories however have a common denominator: the ability to record in some way, design information during a development project in a form

that it can be retrieved, interrogated and manipulated would have a significant contribution to the ability to migrate and enhance future generations of a software product.

Earlier the following definition of experience was introduced:

"Experience is being able to relate a new situation to a situation previously encountered, and, knowing the alternatives and outcomes from the previous situation, being able to deduce the best alternative from the new situation".

This definition of experience, related in this case to human experience, has a similarity to the common theoretical requirement observed from the case studies. If it is possible to practically instantiate the theory, the result would be a software process which had 'built-in' experience. This experience could be passed to future developers or perhaps automatically reused as a form of artificial intelligence within the software process itself.

4.7 Towards a Practical Solution

The remainder of this chapter introduces an additional observation which leads to a theoretical perception of the software process in such a way that a practical approach to maintaining experience can be implemented.

4.7.1 The Pattern of Requirement Changes

An analysis was made of how changes in requirements altered in relation to time. It was observed that the pattern of the impact of user related changes was different from the pattern of the impact of technologically related changes. The patterns of user requirements is illustrated in figure 4.11. The pattern of technological changes is illustrated in figure 4.12. The data used to produce both these illustrations is not scientific but intuitive based upon personal experience.

The Pattern of User Requirements

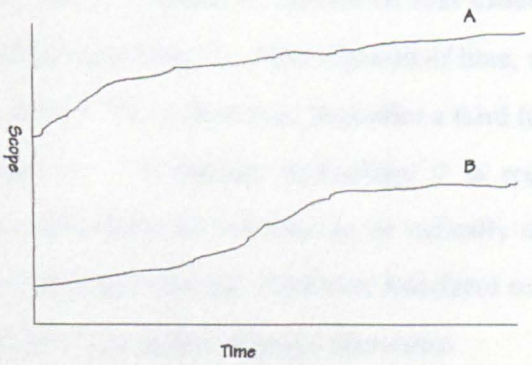

figure 4.11 : A pattern of User related changes

For the purpose of illustration, assume the height of the line A in figure 4.11 represents the scope of all the functions ever included in the user requirements (Requirement For Software). In a similar manner, the height of line B represents the scope of all functions ever removed from the user requirements. The distance between line A and line B therefore represent the actual scope of functional requirements at any instance in time. Time is indicated by the horizontal axis.

The illustration in figure 4.11 indicates that changes in user requirements are regular but individually small.

The Pattern of Technological Changes

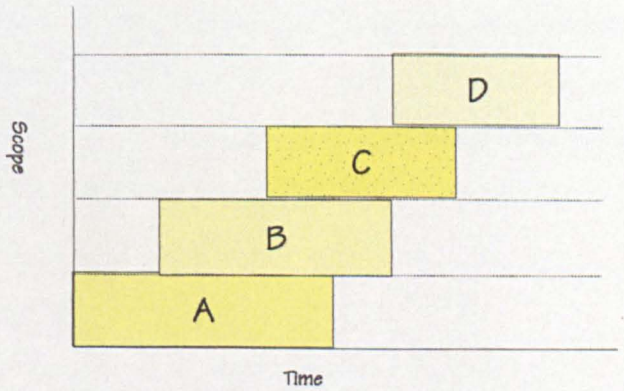

figure 4.12 : A pattern of Technological changes

Figure 4.12 schematically illustrates the Requirements Of Software for the same time period. In this example the system will be required to operate on four different technologies. At the start the system operates using technology A. After a period of time, two different technologies are required in parallel (A and B). For a short time thereafter a third (C) is required followed by a redundancy of technology A. Eventually technology D is required and technology B becomes redundant. The technologies do not have to be radically different. The differences may be as simple as User Language Options, Hardware Interfaces or may be as significant as different platform requirements or program language alterations.

The illustration in figure 4.12 indicates that changes in technological requirements are 'block like' in behaviour.

4.7.2 Using the Pattern to form a Theory

The patterns in figures 4.11 and 4.12 themselves do not contribute to the basis for implementing a practical solution however their production was fundamental in recognising how a solution could be created. Figure 4.13 is a reproduction of figures 4.11 and 4.12 aligned

for a common timescale and a 'slice' S drawn vertically to highlight an arbitrary time point.

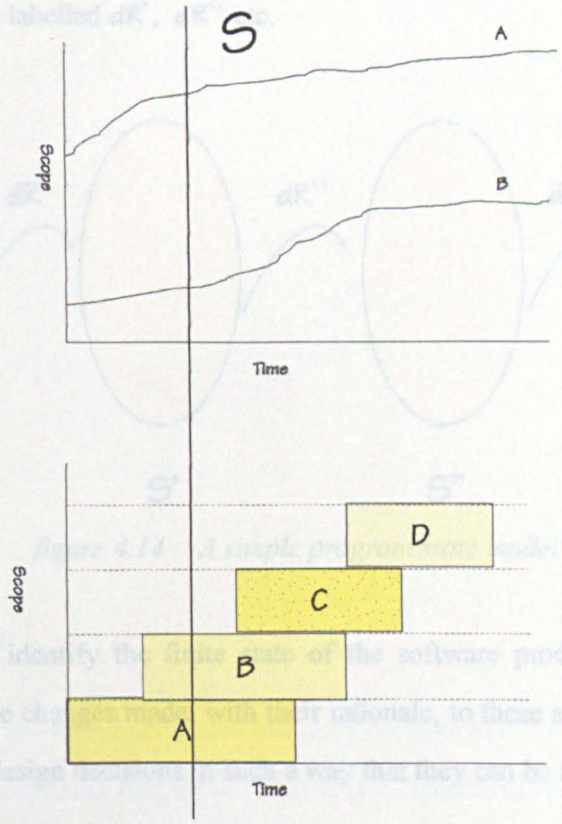

figure 4.13 : A 'slice' of requirements

If it can be assumed that the pattern of requirements is identical to the pattern of the scope of the actual software (albeit displaced by a time element equal to the time taken to implement a set of requirements) then at time S there exists a finite and identifiable state of a software product. At a future time point S` there will be another identifiable state of a software product. The difference between S and S` identifies the set of changes and possibly the set of design decisions which have occurred between time S and S`.

4.8 A Simple Model

A simple model can be created to represent the 'state structure' of a program. This is given in figure 4.14. The state S shown as a vertical line in figure 4.13 is reproduced as an ellipse in figure 4.14. The ellipses S`, S`` and S``` represent states of the program at future times. The

arrows connecting the states indicate the 'set of changes' which represent the transformation between states, they are labelled dR', dR'' etc.

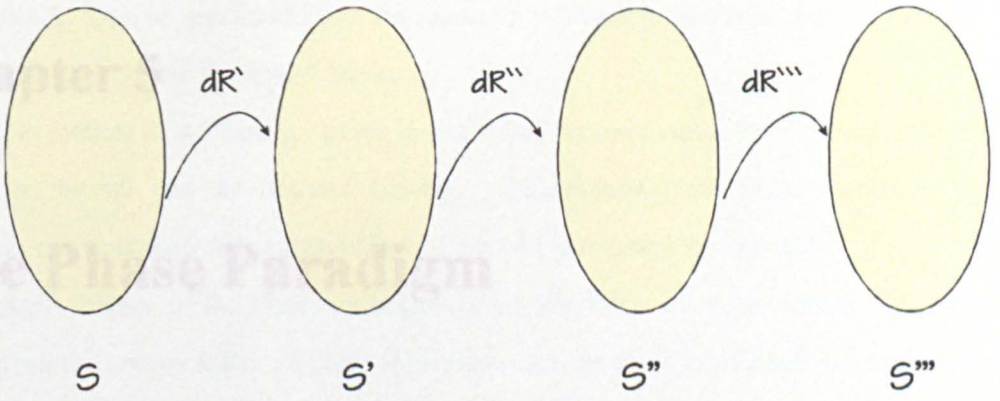

figure 4.14 : A simple program state model

If it is possible to identify the finite state of the software product in terms of tangible attributes and record the changes made, with their rationale, to these attributes then it should be possible to record the design decisions in such a way that they can be reused as experience.

4.9 Conclusion

The requirements for a software process which will help software developers with their attitude towards changing requirements is :

- the ability to define a 'state' of a software product in finite attributes
- the ability to record changes to these states, and the design decisions for these changes
- the ability to retrieve and manipulate this information in a form of 'experience'

One such process is the Phase process.

Chapter 5

The Phase Paradigm

5.1 Introduction

This chapter describes the Phase paradigm and demonstrates how the state of a Phase program can be defined in terms of finite attributes. During this chapter, the following questions will be answered :

- What is a Phase specification ?
- How is Requirements Analysis performed ?
- What is a Phase program?

Structure of this chapter

This chapter begins by clarifying what is meant by Phase software. This includes a discussion on the class of applications intrinsically suitable for developing with the Phase paradigm. 'Screen shots' of a typical Phase program are included to aid visualisation and will help in the understanding of the underlying Phase paradigm.

An integral part of the Phase paradigm is the structure of a Phase program. This section will introduce Phase terminology and demonstrate how the structure can be specified both diagramatically and textually.

At its lowest level, a Phase program is implemented in a target language eg Pascal. In a similar manner to Procedures in a target language, the Phase paradigm uses Procedures for specification. The relationship between the Phase procedures and the target language procedures is described. This in turn introduces the elements of Phase procedures, the definition of which is required to complete the specification.

The execution of a Phase program is controlled by two target language specific elements called the Kernal and the Support Library. Understanding the basic algorithms for these elements will complete the understanding of how a Phase program executes.

A major feature of the Phase paradigm is the repetition of a few simple algorithms. By identifying the commonality of these algorithms and the method of parameterisation it will be shown how the system is suited to the creation of rapid prototypes. Each algorithm can be executed as a prototype by the use of a single command in the small prototype command language.

Documentation is a very important part of any development paradigm and this chapter describes the documentation relevant to (and for the most part, automatically generated by) the Phase paradigm.

This chapter concludes with a summary of the Phase paradigm and explains how a program can be specified as a whole, by specifying the individual elements of a Phase 'state'.

Associated with this chapter is Appendix B, which contains a full description of the way Phase programs are developed, using the Phase process.

5.2 Phase Software

Phase software is software produced using the Phase paradigm. Before discussing its structure or how the design is produced a brief overview is given describing the generic features and user interface. This will place the details of the structure into context.

5.2.1 The Class of Applications

Phase software is not specifically designed for Interactive Business Information Systems (IBIS) applications, but it is with this class of applications that it has been tested and examined. A

description of the features of IBIS software was introduced in chapter 1. To summarise, these applications are :

- Database Oriented
- Human Interactive

A suitable user interface for such applications may have the features :

- 'Form' based for data entry and retrieval
- Menu driven for flow of control

Phase Software incorporates the above user interface features and also :

- Overlapping 'windows' to highlight a 'drill down' detailing of information
- Browse lists to show single-line summaries of information

5.2.2 An Example

An example can be taken showing actual 'screen dumps' of software produced using the Phase CASE tool EDS. This example is a subset of a 'Sales Ledger' program which is designed to track invoices sent to customers and record the payments which are received. This shows the format of:

- Menus
- Browse Screens
- Form Based Data Entry/Retrieval Screens
- Command Line Options
- Overlapping Windows

These examples clarify much of the discussions in the following text.

Menus

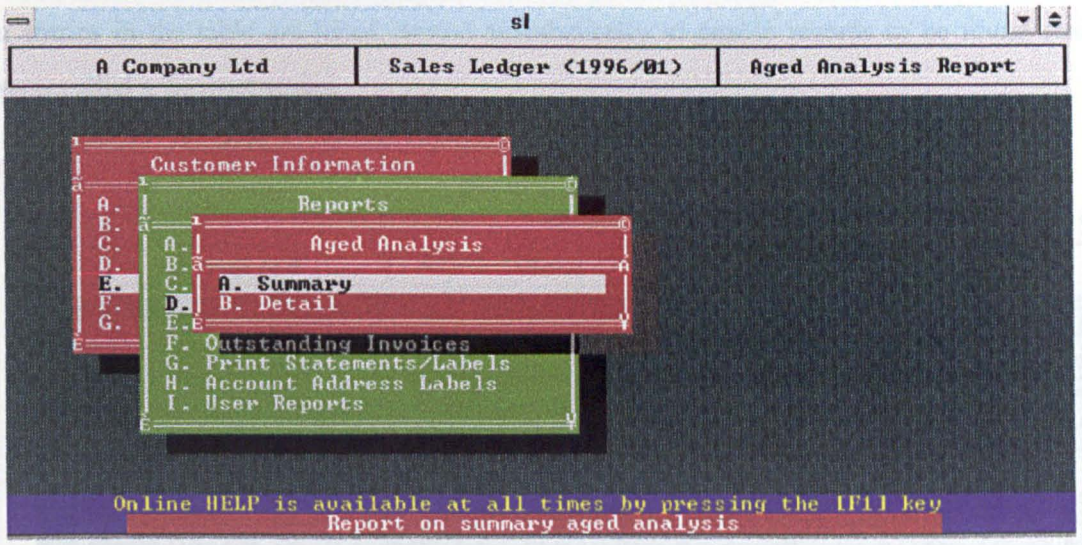

figure 5.1 : A Phase Menu Screen

Menus are hierarchical in structure, nested to any level and with any number of options available.

Browse Screens

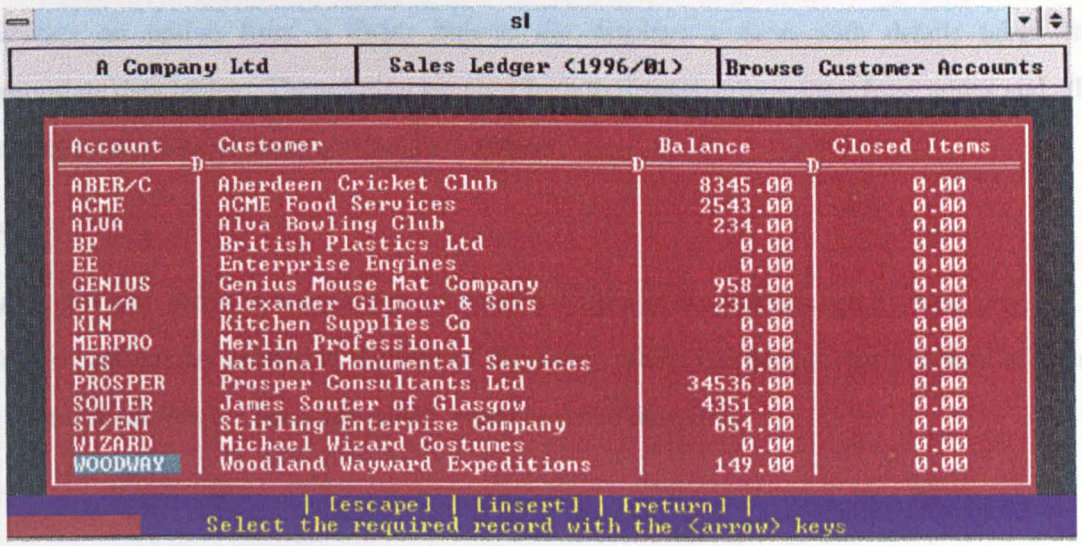

figure 5.2 : A Phase Browse List Screen

Selecting an option from a menu usually requires access to a data table, in Phase programs the entries in the table are listed, sorted alphabetically to enable records to be found easily. This is called a 'browse' screen.

Form Based Data Entry/Retrieval Screens

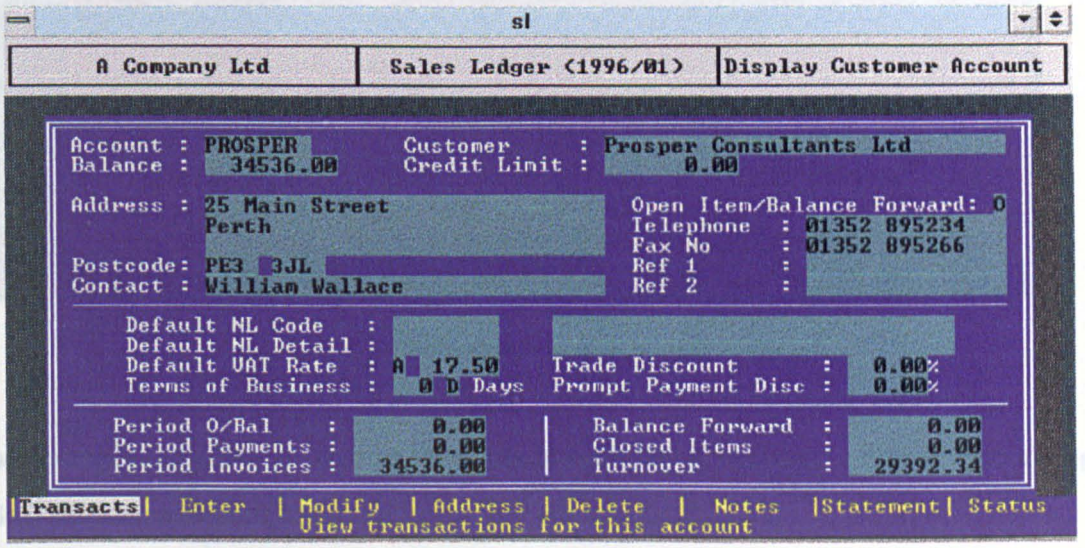

figure 5.3 : A Phase Data Entry/Retrieval Screen

Selecting an option from a Phase browse list displays a form with details taken from appropriate data tables. These forms are used both to display information from tables and allow the user to add data or maintain data in the tables.

Command Line Options

Figure 5.3 also shows a second type of menu for flow of control through a Phase program. A series of options are printed at the foot of the screen which can be selected in a similar manner to a traditional menu. These are called 'command line options'.

Overlapping Windows

In this Sales Ledger example, the first option on the command line is "Transact", this displays an overlapping windows showing a browse list of the invoices and payments making up the balance on the customer account. This is shown in figure 5.4.

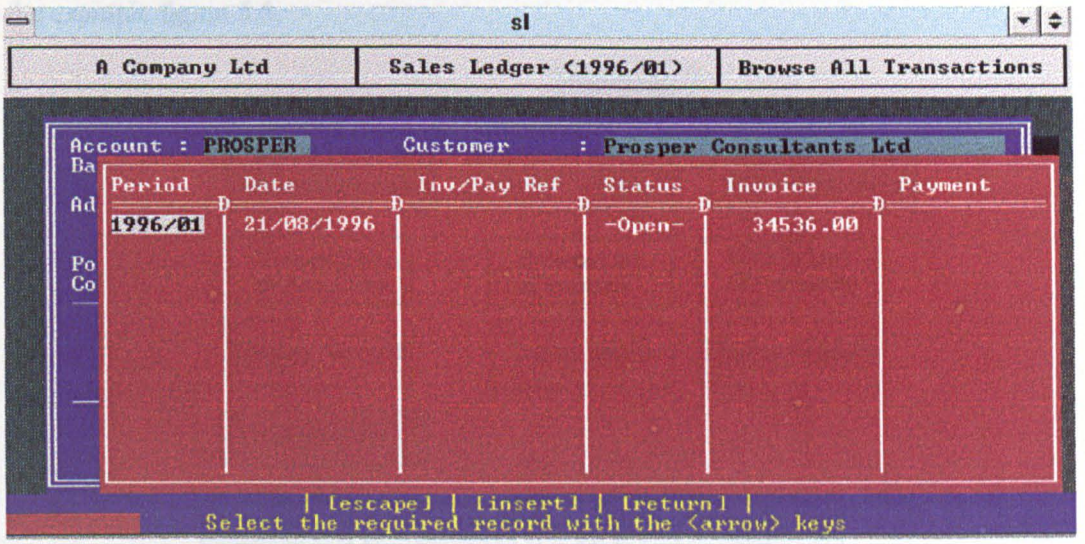

figure 5.4 : A Phase Overlapping Windows Screen

In this example, selecting an invoice from this browse list will display a further overlapping window showing a detailed breakdown of an invoice.

5.3 Phase Software Structure

This process of traversing though a Phase program as Menu, Browse List, Data Form, Command Option, Browse List, Data Form, Command Option etc. becomes an intrinsic part of the software. It is clear to see that this structure can be represented in a directed graph as shown

in the example figure 5.5.

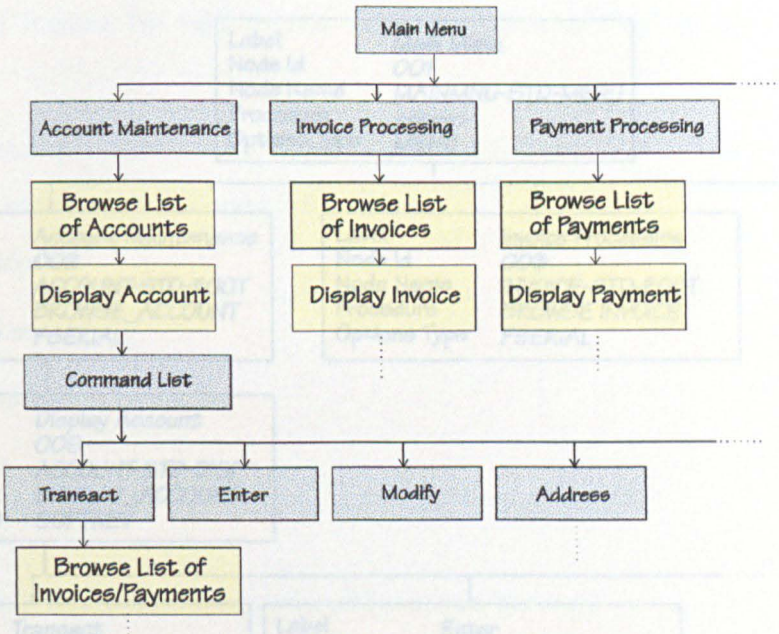

figure 5.5 : The Flow of Control of a Phase program

The diagram in figure 5.5 can be restructured by separating out the actual flow of control from the functionality of a program. If we consider any point in the structure, it can be divided into :

- What functionality do I perform at this point ?
- What options can I select next ?

In Phase terminology the 'functionality performed at this point' is called a procedure and the 'options that can be selected next' is represented as a series of nodes and options. The nodes and options combine to form the flow of control structure. A procedure is 'attached' to a node to specify when it is called. Menu nodes do not have procedures attached. This is illustrated in figure 5.6. Each procedure and node is given a unique reference name and identification number.

5.3.1 The Phase Node Structure

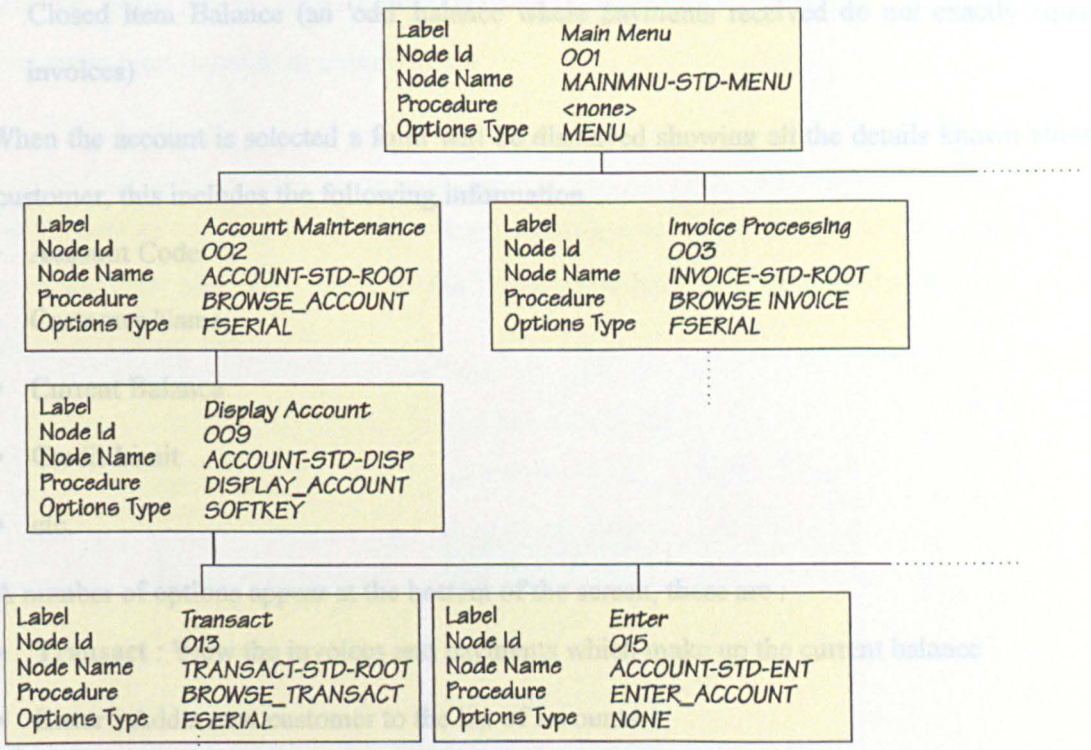

figure 5.6 : A Flow of Control Node Diagram

This structure can also be described :

When the program starts a menu will be displayed. This has the options :

- **Account Maintenance**, where new accounts are set up and enquiries made on existing accounts
- **Invoice Processing**, where invoices to customers are processed
- **Payment Processing**, where payments received from customers are processed
- etc.

Selecting the **Account Maintenance** option will display a list of all customer accounts known to the system, the appropriate account can be selected from the list. The columns shown are :

- Account Code
- Customer Name

- Current Balance
- Closed Item Balance (an 'odd' balance where payments received do not exactly equal invoices)

When the account is selected a form will be displayed showing all the details known about the customer, this includes the following information

- Account Code
- Customer Name
- Current Balance
- Credit Limit
- etc.

A number of options appear at the bottom of the screen, these are :

- **Transact** : View the invoices and payments which make up the current balance
- **Enter** : Add a new customer to the list of accounts
- **Modify** : Amend the current address and credit limit for the account
- etc.

Selecting the **Transact** option will display a list of all the invoices and payments processed for the customer. An invoice or payment can be selected from the list and further details displayed.

etc.

Node Options

There are a limited number of ways that options can be called from a node. These are :

- Menu (as shown in figure 5.1)
- Softkey (as shown in figure 5.3)
- Fserial (described below)
- None (a 'leaf' node with no suboptions.)

The Fserial option is used where there is only 1 option available from a node and this node is selected automatically when the appropriate procedure has finished execution therefore linking procedures together in a serial 'chain'.

5.3.2 The Phase Procedures

Phase procedures are 'blocks' of target language programs which execute as indicated at each node of the flow structure. Each of the Phase procedures will be implemented as a target language procedure call.

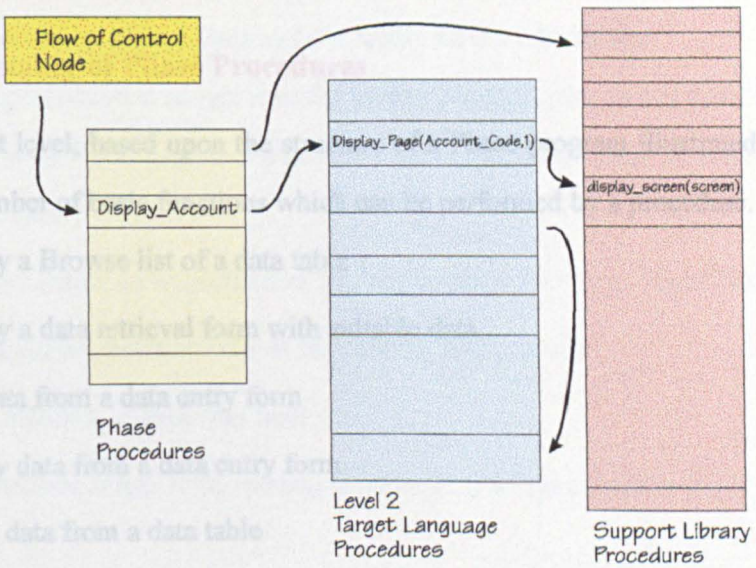

figure 5.7 : Phase Procedures and Target Language Procedures

There are some important issues to raise about these Phase procedures.

- A node may only call one Phase procedure.
- Phase procedures cannot have parameters in the usual sense of formal and actual target language parameters as there is no parameter passing mechanism from nodes.
- Phase procedures cannot call any other Phase procedure
- A Phase procedure may be called from more than one node

- A Phase procedure is completely independent of any other procedure and cannot rely on 'knowing' where it is being called from.
- Where the complexity is such that the functionality cannot be efficiently implemented in a single target procedure, a Phase procedure can call other non-Phase procedures as described in figure 5.7. There additional procedures are called Level 2 procedures and can contain all the characteristics allowed within the target language eg parameter passing mechanism.
- Phase procedures and Level 2 procedures can call upon the services of the Support Library procedures. This is described later in this chapter.

The functionality of Phase Procedures

At an abstract level, based upon the structure of a Phase program illustrated in figure 5.5 there is a finite number of basic functions which can be performed by a procedure. These are :

- Display a Browse list of a data table
- Display a data retrieval form with suitable data
- Add data from a data entry form
- Modify data from a data entry form
- Delete data from a data table
- Print a report
- Perform a batch process

All Phase procedures therefore perform a function from the above list. The "perform a batch process" option also includes any non-standard function from the rest of the list.

5.3.3 Other Phase Entities

Phase procedures have to be more specific than the general descriptions above. This is done in relation to other entities in the Phase structure. These entities are :

- Screens
- Data Items

- Data Tables
- Algorithms

Screens

Screens are the data entry and retrieval forms which display data on the interactive workstations. The consist of two separate parts:

- The 'image'
- The 'field specifications'

The image relates to all the non-field items on the screen including the border, the background colour, graphical lines and the labels for the data fields.

The field specifications are areas of the screen which display actual data from a data table, or where data is input for storage in a data table. Each field has a number of attributes including :

- type of data
- local processing to be performed e.g. Automatic upper case, Right Justify etc.
- data validation functions e.g. is the account code unique
- screen colour to display the field

A screen has one image item and any number of field specification items.

Data Items

Data items are the common link between fields and data tables. Each data item contains a single piece of information. It has a specific type and length.

Data Tables

Data tables relate to the data storage mechanism used by a particular Phase program. These may be 'flat files' or relational databases.

Algorithms

An algorithm is the basic functionality of a procedure. This is identical to the list given earlier in this chapter for example :

A basic algorithm exists to modify data in a data table using a particular screen, it would be in the form (simplified for illustration) :

```
For data table [TABLE] and screen [SCREEN] for key [KEY]
locate in table [TABLE] the data for key [KEY]
match all fields common to [TABLE] and [SCREEN]
display screen [SCREEN]
wait for input
if not escape key
   match all fields common to [SCREEN] and [TABLE]
   store data in [TABLE]
endif
```

The elements in [ ] are parameters.

5.3.4 Procedures Revisited

Procedures exist to collate specific entities together. A procedure will typically have :

- A single algorithm for the basic functionality
- A single screen for data input and/or output
- A set of data items for data transmission
- A set of data tables for data storage

The actual number and type of entities will depend upon the algorithm.

To complete the example given earlier in this chapter assume we have the following elements.

Screen	Data Table	Algorithm	Data items
ACCOUNT	dbACCOUNT	DISPLAY	Account Code
	dbINVOICE	ENTER	Customer Name
		MODIFY	Current Balance
		DELETE	Credit Limit
		BROWSE	Address
			Contact
			Telephone No
			Fax No
			etc.

figure 5.8 : Entities in the example program

The procedure definitions would be :

PROCEDURE : Browse_Account

Screen	Data Table	Algorithm	Data items
	dbACCOUNT	BROWSE	Account Code
			Customer Name
			Current Balance
			Closed Items

PROCEDURE : Display_Account

Screen	Data Table	Algorithm	Data items
ACCOUNT	dbACCOUNT	DISPLAY	Account Code
			Customer Name
			Current Balance
			etc

PROCEDURE : Browse_Transact

Screen	Data Table	Algorithm	Data items
	dbINVOICE	BROWSE	Period
			Date
			Invoice No
			etc

figure 5.9 : Example Procedures

5.4 The Phase Kernel, Support Library & Repository

To summarise, the following types of entities which are maintained in the Phase repository have been introduced :

- Flow of Control Nodes
- Procedures
- Screens
- Data Items
- Data Tables
- Algorithms

Previously, this chapter has described a Phase program and the structure which builds a Phase program. This section describes the remainder of the Phase system which describes how the structure 'works' for an executable program. There are three interrelated items :

- the Phase Kernel
- the Phase Support Library
- the Phase Repository.

The **Phase Kernel** is a library routine which is executed at the start of every Phase program. This routine uses the flow structure maintained as part of the repository to determine the order in which to call Phase procedures.

The **Phase Support Library** is a set of standard functions which are used to interface Phase procedures to entities within the repository during the execution of the program. The Kernel and the Support Library are implemented in the target language.

The Phase Repository is a large database containing all the entities which define a Phase program. Some of these entities (nodes, procedures, screens etc.) have already been discussed, others will be introduced later. The repository is target language independent.

The relationship between the Kernel, Support Library and entities in the repository are shown in figure 5.10.

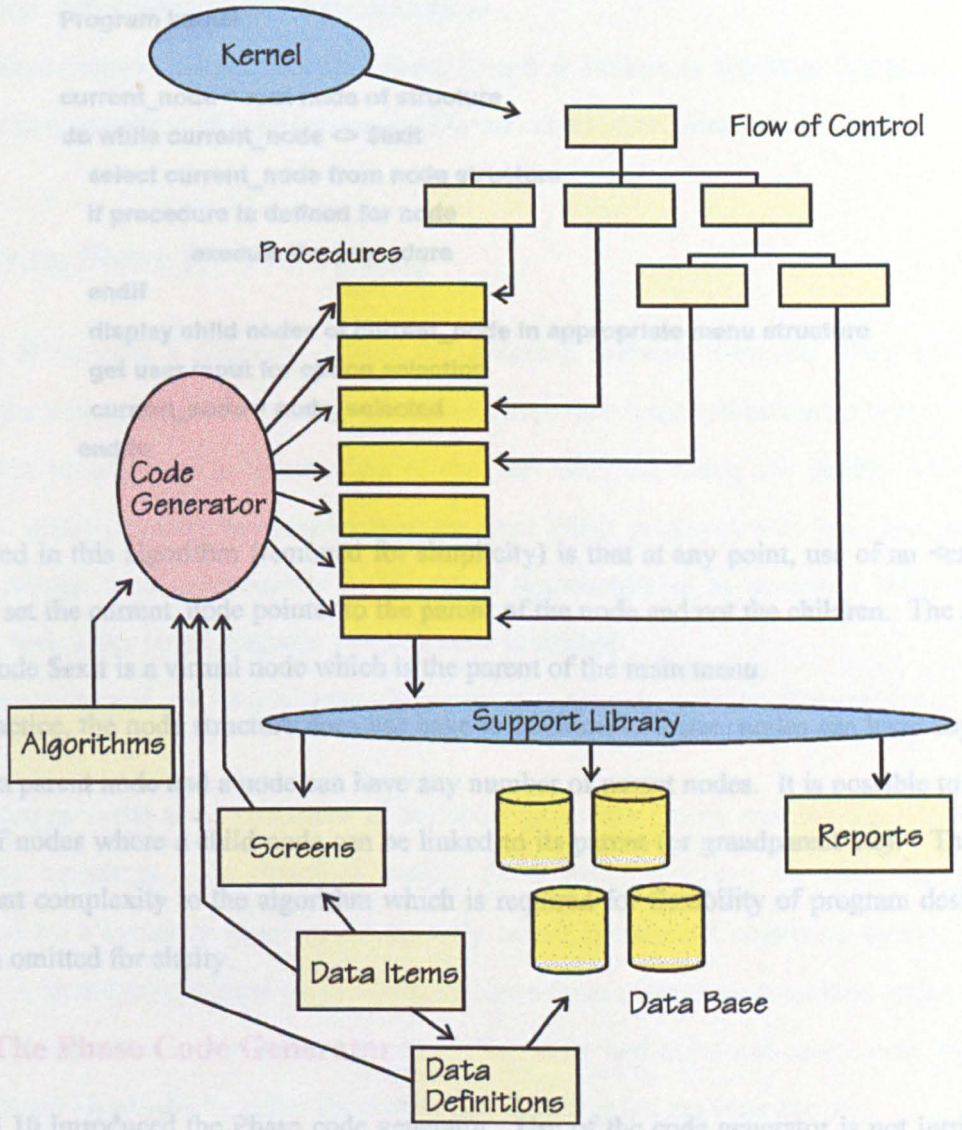

figure 5.10 : The Phase Structure

5.4.1 The Phase Kernel

A Phase program is similar to many other types of program in that it consists of an executable compiled program. The major difference between a Phase program and other programs is that the relationships between procedures are maintained outside the program code, in a repository. In order that this can be achieved, it is vital that the target language supports separately compilable procedures and that procedure names can be determined at run-time.

The kernel is a very simple program with an algorithm (suitably simplified) as shown below:

```
Program kernel

current_node = root node of structure
do while current_node <> $exit
   select current_node from node structure
   if procedure is defined for node
         execute the procedure
   endif
   display child nodes of current_node in appropriate menu structure
   get user input for option selection
   current_node = node_selected
enddo
```

Implied in this algorithm (removed for simplicity) is that at any point, use of an **<escape>** key will set the current_node pointer to the parent of the node and not the children. The special pseudonode **$exit** is a virtual node which is the parent of the main menu.

In practice, the node structure does not have to conform to a tree, nodes can have any other node as a parent node and a node can have any number of parent nodes. It is possible to have a 'cycle' of nodes where a child node can be linked to its parent (or grandparent etc). This adds significant complexity to the algorithm which is required for flexibility of program design but has been omitted for clarity.

5.4.2 The Phase Code Generator

Figure 5.10 introduced the Phase code generator. Use of the code generator is not intrinsic to the structure of a Phase program but exists as a productivity tool for the developer. The

commonality of program functions and the information available with the repository entity is an ideal candidate for automatic code generation. The parameterisation of the algorithms as shown in the example earlier in this chapter allows target code to be created by simple substitution similar to the technique found on many macro assemblers.

5.4.3 The Phase Support Library

The Phase Support Library exists as a set of functions which allow application procedures to access elements within the repository at run-time. The main use of this is the access required for the screen definitions and data table definitions.

The Phase Support Library, like the Phase Kernel is written in the target language. This allows for easy migration of applications to different database and screen technologies.

5.5 Using Phase for Prototyping

Appendix B provides a complete process for designing software using the Phase paradigm, however the principles of Phase prototyping are an important issue and discussed below.

A Phase prototype is a construction of the user interface using the entities within the repository which demonstrates exactly how the final Phase programs will 'look and feel'. A Phase prototype (and consequently a full set of documentation as described later) can be produced before any target language program code is created.

A Phase prototype and its associated documentation provide an 'as built' specification of the design.

A prototype exists as an 'execution' of the node, procedure, screen and data-item definitions within the repository. As described earlier, for a Phase program, the node structure is interpreted by a kernel program to dynamically create menu's and command options during runtime. A 'prototyping' kernel exists which recreates these menus and command options using an identical algorithm to the finished product. The menu and command options can therefore be reproduced identically by either the prototyping kernel or the run-time kernel.

The remainder of the user interface is made up by screen definitions and browse definitions, both of these entities are defined within the repository, however the problem exists to determine which screens and browse definitions to use at each point.

A finished program uses the Phase procedure definitions to perform the required functionality in the form of executable target language code, these procedure definitions provide a suitable mechanism for defining a prototype.

Earlier in this chapter it was stated that procedures can perform according to only a small set of functions. Most of these functions involve a single screen (or browse definition). These functions can be simulated for prototyping purposes by using a prototyping command set of only four commands with a minimum number of parameters. These commands are :

- DISPLAY "<screen definition> <version>"
- BROWSE "<browse definition>"
- REPORT "<filename>"
- MESSAGE "<message>"

5.5.1 The Phase Prototype Specification Language

Display "<screen> < version>"

This command allows the Prototype Executor to fetch the <screen> definition from the repository and display it on the workstation. A screen definition at this point may only consist of the 'image'. In 'early' prototypes, field definitions may be substituted on the image in a similar manner to graphics characters or field labels.

If, however, field definitions are included, data input and retrieval can be simulated during prototype execution. This reaction of this, depends upon the second parameter for the command. This is the 'version' parameter. Each screen can have up to 10 different 'versions' numbered 1 to 10. Each version applies to the combination of fields which are 'read only' and 'data input'. For example, version 1 of a screen may have all the fields marked as 'read only' and used in a 'display' type algorithm. Version 2 may have all the fields available for data input and used for the creation of a new record using the 'enter' algorithm. Version 3 may have all

the fields available for data input except the key fields (which cannot be changed once the record has been created) etc.

At this stage there is no concept of a data table. For simulation purposes, each field definition has a 'dummy data' field into which any data input is stored. This provides a very realistic method of simulating data entry.

Browse "<browse definition>"

This command simulates a browse screen using column headings defined within the repository. For these screens there is no data available and all the columns appear blank. Although this means that the simulation of the prototype gives a slightly different screen from the final versions, the difference is insignificant.

Report "<filename>"

This command simply takes the <filename>, extracts the definition from the repository and copies the information without modification to the print device. The <filename> has been previously created as a sample report using a standard text editor. The data on the report obviously has not been derived from anywhere in the system but simply 'typed in'.

Message "<Message>"

This option does little more that display the message on a "status" information line on the screen. Execution then suspends for a given time period (say 5 seconds) and then control is returned to the Prototype Executor to continue processing. This delay represents a process being executed (although in reality nothing is done at all).

These four functions, together with the menu definitions allow for an extremely useful prototype to be created and executed. Whilst being extremely simple in operation it provides the user with the look and feel (including relevant pauses for process execution) of the final application. This is the fundamental principle for Rapid Prototyping.

5.6 Phase and Documentation

The Phase prototype is the primary means of communication between users and developers. The prototype, however, requires a computer for execution. There are additional benefits to be gained by providing 'hardcopy' documentation which can be 'reviewed at leisure' and ideal for annotating with comments. These annotations can be entered subsequently into the repository using the appropriate editors within the system.

The Phase case tools provide a number of representations of the prototype. These are:

- Flow of control 'tree' structure
- Entity Relationship Diagram
- Hardcopy prototype.
- Database Structure
- Technical Reference Manual

Flow of Control 'Tree' Structure

This is an automatically produced diagram similar to figure 5.6 listing all the interconnections of nodes in the form of a directed graph. It displays a high level overview of a software design.

Entity Relationship Diagram

This is an automatically produced diagram which prints for selected entities within the repository showing all the hyperlink type connections to associated entities. This is particularly useful for finding the 'consequences' of change.

Hardcopy Prototype

This document prints a single A4 page for each node in the system. This can be used in conjunction with the structure 'tree' diagram described above. Printed on each page is a 'screen dump' of the screen which would be displayed during the execution of a prototype. At the foot of the screen the options are listed and a full cross reference is made to the appropriate page numbers. This document is ideal for use when reviewing a prototype as comments can be noted within their context.

Database Structure

A traditional 'data dictionary' type report listing the structure and relationship of all the data tables.

Technical Reference Manual

The technical reference manual is a document structured automatically by the flow of control structure of the Phase prototype. It contains an appropriate selection of 'screen dumps' complete with automatically generated comments taken from notes maintained with entities within the repository. The is the basis for a user reference manual which can be distributed with the final product.

5.7 A Summary of the Phase Environment

The term 'Phase Environment' is used as a generic term for entities within the Phase paradigm which includes the features of the supporting CASE tools. This section lists all the features of these tools for completeness as reference is made to them in later chapters.

- Entity Editors
  - Flow of Control Node Editor
  - Procedure Editor
  - Screen Painter and Editor
  - Database Dictionary Editor
  - Data Item Editor
  - Algorithm Editor
- Generators
  - Program Code Generator
  - Screen Definition Code Generator
  - 'Reference Manual' Documentation Generator
  - Database Generator
- Project Management Features
  - Entity Modification Log

- Request for Program Update (RPU List) "Wish List"
- Completion Statistics and Status Reports
- Node Structure Diagram
- Entity Relationship Diagram

- Other
  - Prototype Executor

5.8 Phase and the Process State Model

At the end of chapter 4, a simple program state model was introduced with three preconditions for helping developers with their attitude towards change. These are reproduced here for clarity.

The requirements for a software process which will help software developers with their attitude towards changing requirements are :

- the ability to define a 'state' of a software product in finite attributes
- the ability to record changes to these states, and the design decisions for these changes
- the ability to retrieve and manipulate this information in a form of 'experience'

This chapter has described part of the Phase paradigm, the remainder of this chapter will show how this information presented so far relates to the first precondition above.

5.8.1 The Definition of a Program State

The elements of the Phase repository have been presented. They are listed in figure 5.11 below together with symbols which will be used in later chapters.

Element	Symbol
Flow of Control Node	N
Data Item	I
Screens	S
Data Table	D
Algorithm	A
Procedure	P

figure 5.11 : The elements of a Phase Repository

Each entity can be individually identified with a unique identifier and each type of entity has a finite set of attributes. A complete set of attributes is contained in Appendix A. The state S of figure 4.14 can now be represented by the set of all the Phase entities defined for a particular program as shown in figure 5.12.

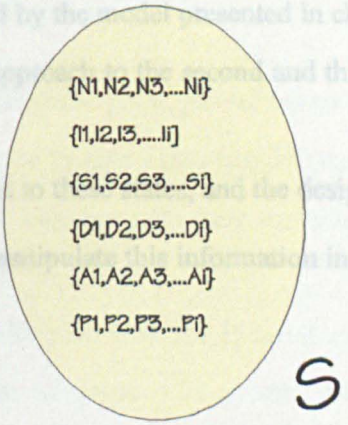

figure 5.12 : The components of a Phase state

These elements do not exist in isolation to one another but form a complex hierarchical interrelationship. Each element can be related according to the following rules:

N -> {N} P

P -> A {D} {S}

D -> {D} {I}

S -> {I}

The symbol -> means "is hierarchically related to"

The symbol {} means "any number of". The absence of {} indicates that only a single relationship can exist between a discrete entity of these types.

5.9 Conclusion

The Phase paradigm uses a program structure which relies upon the definition of a set of entities within a repository. These definitions can be 'executed' as a prototype to allow a user review of the software before any application code is created. When final programs are required, these same definitions can be used by the automatic code generators within the Phase CASE tools to aid programmer productivity. The definitions can also be provided in 'hard-copy' form.

This repository structure of a program definition makes it possible to tangibly represent a program state in the form indicated by the model presented in chapter 4.

Chapter 6 provides the Phase approach to the second and third preconditions attached to the model, namely,

- The ability to record changes to these states, and the design decisions for these changes
- The ability to retrieve and manipulate this information in a form of 'experience'

Chapter 6

Defining and Reusing Phase Experience

6.1 Introduction

In chapter 5 , it was shown how the Phase paradigm relates to the model described in chapter 4 by providing a breakdown of a Phase program into tangible components or entities. The set of all the entities defined at any time, together with their attributes is represented as a 'State' $\mathcal{S}$ in the illustration given in figure 4.14.

The aim of this chapter is to describe how the Phase paradigm relates to the changing of states, indicated in the illustration of figure 4.14 as $\mathcal{dR}$, with the purpose of being able to capture 'design experience' in a form which can be reused.

This chapter is split into three sections:

- A history of how the information $\mathcal{dR}$ was captured with accuracy
- A discussion on how the information can be retrieved and manipulated
- A discussion on how the information can be analysed to provide 'experience'

6.2 The Principle of Recording Design Changes

Recording changes to entities within a repository is obviously easy provided each editor that is used to physically edit an entity can provide suitable information to a 'log' file each time an edit

is performed. The use of 'edit' in this sense also includes the creation of new entities and the deletion of entities. What is not so obvious is what information should be logged and how should this information be structured.

There are a number of observations which can be made :

- Information to be logged should be obtained 'automatically' and not require manual input as 'human nature' will bypass manual input under pressure of time.
- The log should be maintained unobtrusively to prevent the logging interfering with the productivity of the development
- The log should be as 'space efficient' as possible, as the number of entries will be large
- The information logged should refer to why information is changed as well as what information has changed.

6.3 When is a State, Not a State?

In the illustration in figure 4.14, two development states are separated by a 'change in state'. The question is asked :

What determines when a program is 'in a state', and can a program ever be 'between states'?

Strictly speaking, every change which is made to an entity alters the state of the program. Consider the circumstance that a change to an algorithm takes four attempts by a designer before it correctly reflects a concept. Each of the first three steps were simply 'bad' attempts and intermediary. They did not reflect design decision changes but simply the correcting of errors. In this instance it is proposed not to 'recognise' these intermediate stages.

Consider also, where a single design decision may require a number of entities to be altered. For example a decision may be taken to remove the concept of a 'telex' field from a contact database (due the redundancy of telex machines over fax and email). This would almost certainly affect :

- The data table where the telex field stored the data

- The screen where the text field displayed the data
- The redundancy of the data item 'telex'
- Any procedure which referred to the data item 'telex'

This would require a minimum of four edits in the repository, with a possible four changes in program states. In this instance it is also proposed that only one change in state is recognised.

In summary, it is possible that a program can be between states and that any number of entity changes can be made between states. In reality, during these 'inbetween' states, the program can be considered 'unstable' (as it almost definitely would not execute correctly due to inconsistencies between entities). Consequently a state can be redefined as a point in time where the program is stable (not unstable) and a set of related changes are considered 'complete'.

6.4 The Example Data in this Chapter

For the purposes of illustration, the remainder of this chapter uses examples taken from a Phase project, the Sales Ledger application introduced in chapter 5. A brief introduction to this application will place these examples in context.

The Sales Ledger application was first started in 1990, this was one of the first applications to be developed using the EDS CASE tool. At this time there were no logging facilities enabled. This unfortunately means that there is no early design history available. This is not detrimental to the examples.

This application is one of a suite of core programs for general business administration. It integrates fully with the other programs in the suite and consequently has to 'know' about external applications.

The 'size' of the application can be indicated by the number of entities in the repository. At 1996 these are shown in figure 6.1.

Element Type	No of Items
Flow of Control Nodes	163
Data Items	283
Screens	25
Data Tables	12
Algorithms	134
Procedures	246

figure 6.1 : The 'size' of the example application

There are presently over 5000 log entries relating to changes made to individual entities recorded in the period January 1991 to January 1996.

There are 28 commercial installations of the application with a total of over 40 users. Many installations are networked on networks with a 50 user capacity.

6.5 The Phase History of Change Recording Development

The methods by which change information was recorded by the Phase CASE tools altered four times over the period of study. Each new method of recording was prompted by a lack of rigour in the existing method, unreliable data does not result in reliable analysis. These four methods of recording information, each one progressive, are presented in sequence. This provides not only a justification for the final method, but provides an insight into the 'design decisions' which were taken along the way.

In the reading of this chapter, by presenting the 'design decisions', the experience which was learnt by me, the designer, will be passed to you, the reader. This is a practical demonstration of the principle of 'experience passing' which is being proposed in this thesis.

6.5.1 Recording Design Changes : A First Attempt

A 'log' file was added to the EDS CASE tool in 1991. Figure 6.2 represents the structure of the file:

Field	Description
Date	The date a change was made
Time	The time a change was made
Username	The UserId of the person making the change
Entity_Type	Type of Entity eg Node, Screen, Data Table etc
Entity_Id	The unique identifier for the entity
Remark	The type of change made eg Created,Modified,Deleted etc

figure 6.2 : The structure of a simple log file

Each entity editor in EDS was altered to create an entry in the log file whenever a change was made to a repository entity. This satisfied three of the four observations made about logging earlier in this chapter. It did not include any why information.

The method of logging changes was active from January 1991 to the end of December 1991. This very basic form of history logging simply showed which entities were being created or amended and by whom.

At this early stage a significant amount of management information could be extracted. This included :

- A definitive record of when entities were changed. This helped 'debugging' by knowing which entities had been changed 'recently' around the time that a 'bug' had been first noticed.
- An indisputable record of who changed entities. This put an end to the common 'I didn't touch it' comments from developers.
- An indication of the amount of time expended on a project which could be translated into a cost for a project.
- A high level 'activity' graph could be produced which gives a clear 'picture' of when changes were made. An example is given in figure 6.3.

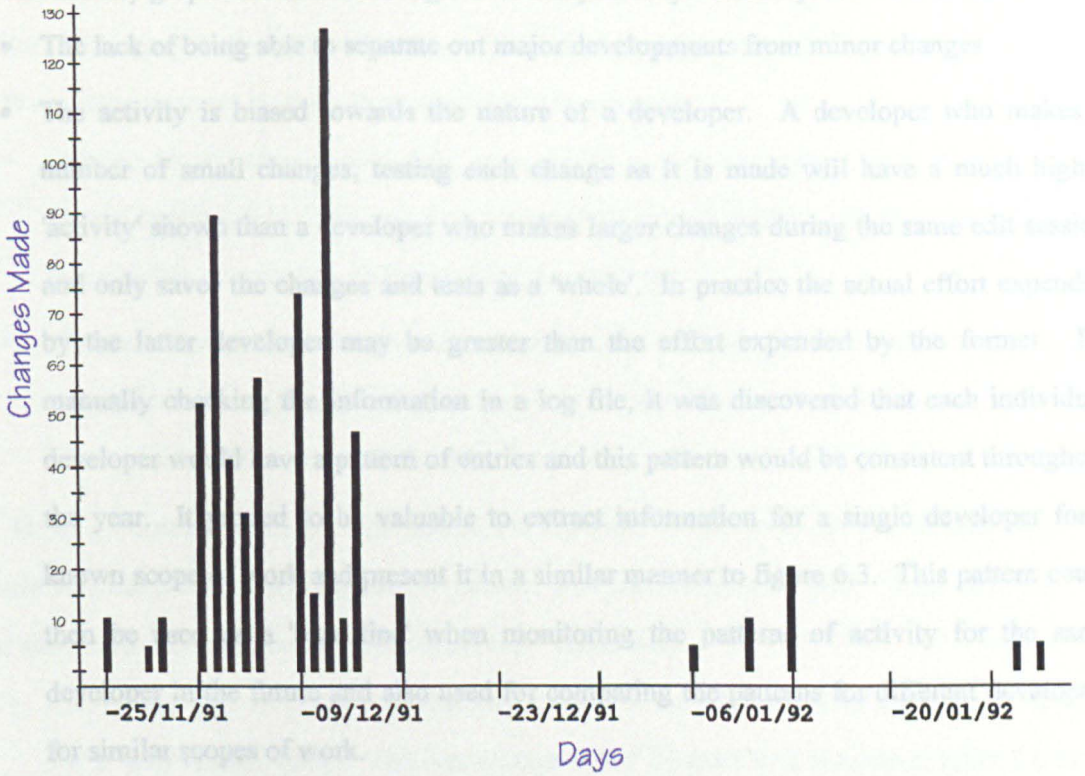

figure 6.3 : General activity graph for a development

This activity graph shows the number of changes (the vertical axis) made to entities on each day (the horizontal axis). This indicates that during the period end of November / beginning of December there was a peak of activity. This was followed by a period of no activity (due in this instance to the Christmas holidays).

By examining a number of these activity graphs, a pattern emerges.

- A block of activity appears around the time that a new installation takes place. This implies that as new users obtain the software, new concepts are introduced (or existing concepts altered). In the above example, a new installation was due in the middle of January.
- A block of heavy activity is often followed by a tailoring of activity. This is indicative of final debugging where the rate of changes slow down. In the example, this is shown in the first part of the graph

The activity graphs are useful for high level analysis only. The major criticisms are :

- The lack of being able to separate out major developments from minor changes
- The activity is biased towards the nature of a developer. A developer who makes a number of small changes, testing each change as it is made will have a much higher 'activity' shown than a developer who makes larger changes during the same edit session and only saves the changes and tests as a 'whole'. In practice the actual effort expended by the latter developer may be greater than the effort expended by the former. By manually checking the information in a log file, it was discovered that each individual developer would have a pattern of entries and this pattern would be consistent throughout the year. It proved to be valuable to extract information for a single developer for a known scope of work and present it in a similar manner to figure 6.3. This pattern could then be used as a 'base line' when monitoring the patterns of activity for the same developer in the future and also used for comparing the patterns for different developers for similar scopes of work.

6.5.2 Recording Design Changes : Adding the "why" Question

It was during January 1992 that additions where made to EDS to include information relating to 'why' entries were modified. This was done by manually creating an entry in a second data table with a structure indicated in figure 6.4. This entry contained an identification number and a textual description.

Field	Description
RPU_Ref	A unique Reference Number
Description	A text field containing information relating to 'why' changes occur. This is simple free-format notes

figure 6.4 : The structure of the 'why' table

The RPU_Ref was built on the acronym introduced to identify a 'why' entry called a Request for Program Update (RPU). The format of an RPU_Ref was a 2 digit mnemonic representing

the application and a numeric sequence number starting at the number 1001 e.g. SL/1234 would be the 234th entry in the 'why' table for the Sales Ledger application.

Procedures were set in place along the following concepts:

- Changes to programs should only be made for a common reason. For example, when adding features, add them one at a time and change all entities in the repository which relate to this new feature, before starting the next one.
- When all changes have been made, create an entry in the 'why' table containing a textual description of the change. At this point an entry will also be made in the history log which 'date and time stamps' the 'why' table entry.

These procedures were left operational for a period of one year. It was decided that this would be a minimum period required to allow a 'true' picture to be formed. Three benefits, over and above the previous benefits were observed. These were :

- The reason why an entity was changed could now be determined by scanning the log file for a 'why reason' entry. This knowledge could be used in a manner described later in this chapter.
- Each change of state could now be identified using the unique RPU_ref field in the 'why' table.
- Developers were now required to explicitly 'finish' a development exercise. Forcing this issue had the benefit that it removed another common developers phrase - "its about 95% complete". If any entry appears in the 'why' table then it was complete. If no entry appears than it was not complete.

An example of the data recorded is given in figure 6.5.

Log Table

Date	Time	Userid	Entity_Type	Entity_id	Remark
10/01/92	10:34	ALAN	SCREEN	ACCOUNT	Item TELEX Removed
10/01/92	10:36	ALAN	TABLE	dbACCOUNT	Item TELEX Removed
10/01/92	10:37	ALAN	PROCEDURE	DISPLAY_ACCOUNT	Modified
10/01/92	10:42	ALAN	PROCEDURE	MODIFY_ACCOUNT	Modified
10/01/92	10:46	ALAN	PROCEDURE	ENTER_ACCOUNT	Modified

10/01/92	10:52	ALAN	DATA ITEM	TELEX	Deleted
10/01/92	11:36	ALAN	RPU	SL/1234	Released

Why Table

RPU_Ref	Description
SL/1234	Remove the concept of a telex number from the customer account as this information is no longer available.

figure 6.5 : Example Data Recorded

In practice, the data collected was not acceptable for the following reasons :

- Concurrent development by more than one developer meant that entries in the log file from two or more developers were mixed together. To search for the 'why' reason, entries created by other developers had to be ignored.
- Developers would be tempted to 'fix small bugs' at the same time that major development was being done. This meant that the reasons why an entity, which was changed under the 'small bug' heading, would be lost and replaced with the 'major development' heading.
- Programmers had a resistance to marking work as complete, perhaps until further testing was completed. This meant that entries were not always created in the 'why' file at the correct time. This meant that changes for the 'next' step in the development were often included in the 'current' step in the development.
- Functional changes being made to programs varied dramatically in size. Some major functional changes would take days, perhaps weeks. Others would be simple changes taking perhaps minutes to complete. Timing became an important issue. If a major development exercise was being performed, small changes realistically had to wait until the 'why' entries for the major development had been completed
- There was no secure method of 'policing' the data to ensure accuracy and consistency of adherence to the day-to-day procedures. Analysis of the log files was periodically performed manually to determine their accuracy. The development team would be a mixture of mature skilled personnel and new junior personnel. The analysis showed that in general, the junior members would not mix changes for different reasons as often as

the senior members, primarily because they were trained in these procedures from the start and also because they had smaller and more defined tasks to complete.

6.5.3 Recording Design Changes : Retiming the "why" Question

Direct action was required to improve the accuracy of the data collected before any attempt could be made to use it for extracting 'design' information. The major issue was the problem of concurrent development, either by different members of the development team or by a single member working on more than one functional aspect at a time.

A feature being added to the EDS CASE tool at the start of 1993 was a 'Wish List' concept, a common feature within program development environments. This wish list would record requests for program changes in a central data table to allow a methodical means of logging requests and reviewing possible changes at management design meetings. The structure of the wish list table was seen as a superset of the 'why' table previously introduced, adding fields for :

- who requested the change
- when it was requested
- the benefit that the change would bring to the use of the program
- etc.

As this wish list was introduced, a simple addition and restructuring of the 'start-up' sequence of EDS enforced the selection of an entry from the wish list before any changes could be made to entities in the repository. This effectively meant that the 'why' question was asked before any changes were being made replacing the previous system of asking the why question after changes were made. The identifier for the 'why' question would be 'remembered' by the CASE tool throughout the session. If no relevant entry was in the wish list, the developer would be able to add a suitable entry before proceeding. This virtually eliminated any risk of 'cheating' caused by selecting 'any old entry' just to be allowed to make the change.

In addition, the RPU_Ref field was also added to the history log file and the 'why' identifier recorded each time an entry was written to the log.

The following improvements were observed :

- Concurrent development could easily be accommodated as each developer could select his/her own RPU to develop.
- It would be possible for more than one developer to work on the same RPU at the same time (editing different entities) where, for example, a new feature being added was sufficiently large or urgent for multiple developers to be cost effective.
- It was no longer necessary to complete an RPU before starting a new one. This meant that small 'bug' fixes could be given their own RPU number and the changes made under this number whilst a larger development exercise was still proceeding.
- It was still necessary to mark work done under an RPU as complete from a management control point of view. This was done by changing the 'status' of the wish list entry and allocating a 'release number'. This meant that 'states' were given a different numbering system for identification purposes.

Data was collected in this way until the middle of 1994 when a further check on the accuracy was performed. The result at this time was significantly improved. The system was less 'stressful' to use as it was more intuitive to select and document 'reasons for change' at the start of a session than at the end.

Other side benefits were noticed :

- Forcing programmers to document changes before they happened improved the general efficiency of the development. It forced them to 'think through' the change before it happened and prompted questions regarding the consequences of the change.
- It was now easily identifiable when a system was 'unstable' as it was represented by a list of RPU's which had been started but not marked as complete. This gave reassurance and additional control during the release of systems to users.
- Delegation of work to junior staff was easier as it was possible to list all the changes relating to a specific RPU. This made it possible to 'police' changes which were being made.

It was this third benefit which prompted a further refinement to the data collection exercise. This is described below.

6.5.4 Recording Design Changes : Quality Inspection Documentation

After analysing the data in June 1994, the data being collected was about 85% accurate. Accuracy in this case relates purely to the tagging of changes of entities to correct 'why' reasons and was calculated by retrospectively manually checking each entry in the log file against its 'why' reason. This level of accuracy could have been regarded as sufficient but was time-consuming to police. In order to improve the 'policing' the following refinement was made. This refinement takes the accuracy to over 98% and is in use to the present day.

Working on the observations that :

- the system knows about every change made 'under an RPU'
- a 'why' reason has been given for the RPU before changes are made
- each application (Phase program) will have a development staff member who is 'experienced' in the application, either because he/she was part of the original development team or has been involved with development for a period of time. This person is referred to as the 'application supervisor' and has overall responsibility for the quality of the application
- generally it requires 'quality staff' to create a quality software product, but being able to use junior developers effectively without compromising quality would provide a cost benefit.

the formation of a 'Quality Inspection Record' (QIR) was introduced. A QIR is a form produced by the EDS CASE tool before an RPU could be marked as released. An example is show in figure 6.6.

The QIR is designed as a mechanism for policing changes made to a program repository. It consists of three parts :

- The information relating to the 'why' information contained in an RPU
- A list of all the entities which have changed, marked with the RPU reference. This is a summary from the history log file
- A signature box

Application : Sales Ledger — QUALITY INSPECTION RECORD - 19/04/1996 — Page No : 1

Request : 1312 Add additional contacts to the site address file

Notes : .H1.Enhancement

In order to increase the usability of the site file, three additional contacts fields have been added. Note that these fields are not accessed by another module at this time.

This requires utility *** USLC0315 ***

Action : JOHN

Source : PHIL	Date : 20/12/1995	Release : C.03.15
Priority : 2	Status : C	

Type	Entity	Action/Remark	User	Checked	QC
Database	SLADDRSS	Added CONTACT2 Added CONTACT3 Added CONTACT4	PHIL PHIL PHIL		
Item	CONTACT2	Entered Modified	PHIL PHIL		
	CONTACT3	Entered Modified	PHIL PHIL		
	CONTACT4	Entered Modified	PHIL PHIL		
Macro	BROWSE_ADDRESS	Edited File	PHIL		
	DISPLAY_ADDRESS	Edited File	PHIL		
	ENTER_ADDRESS	Edited File	PHIL		
	MODIFY_ADDRESS	Edited File	PHIL		
Screen	ADDRESS	Saved	PHIL		
√ RPU	C.03.15	Released	PHIL		

Authorised to Release : _____

QC Signature : _____

figure 6.6 : Example Quality Inspection Record

The QIR Request Information

At a time before the QIR, the 'why' information was held as unstructured text. The QIR has a more structured format for the information which both:

- encourages the information to be entered
- makes 'computer' analysis of the information easier

Briefly, the information maintained is :

- The original request information. This is a short description, often in developers note form summarising the changes required
- Technical notes and areas for consideration. This is a free format text note which requires information from an 'experienced' developer to indicate potential problems and conflicts. It is this 'experience' which it is hoped may eventually be available direct from the Phase process itself.
- User information. This is the information which can be given to users as release notes to inform them of the changes made. This description should be 'untechnical'.
- Source, Priority, Date Raised, Status and Action : A number of management information fields used for presenting and analysing request information.

The QIR Change Log Information

The second part of the QIR lists all the entities which have been directly altered (or created, or deleted) as part of this set of changes. This is pre-sorted by entity type and indicates

- The name of the entity
- The type of change made (added, modified, deleted)
- The name of the developer who made the change

An entry will appear on the list for every combination of these. For example, if an entity is modified by two different developers then the names of both developers will be listed. However if an entity is modified a number of times by the same developer then it only appears once.

On the right hand side of the QIR are two blank columns headed

- Checked
- Quality Control (QC)

These columns are used as described below.

Using the QIR

The QIR is used as follows:

- When a developer (Junior or Senior) decides that an RPU is ready for release, the QIR is printed.
- The form is handed to the application supervisor (who may actually be the same developer)
- The application supervisor looks at each entry on the form and makes a judgement based upon :
  - The complexity of the changes required
  - The ability of the person making the change
- The judgement is made to either check the work done or to accept that it has been done properly (it should already have been tested by this time). If the change to an entity is assumed correct then the "Checked" box on the form opposite the entity should be 'ticked'. If the change is physically checked (using a visual inspection) then the box is 'initialled'.
- The person checking will also be able to establish whether
  - Entities have been changed which do not correspond to the description
  - Entities have not been changed which should have been
- If for any reason the person checking is not satisfied with the changes, the form will be returned to the developer for rework. If everything is in order then it will be signed in the "Authorised to Release" box. At this stage the RPU can be "Released". The release number, which was allocated by the development system is then written onto the QIR and filed.

This checking mechanism can also be policed. This policing can be done by any other developer (not necessarily a senior member). It is policed by checking the judgement of the

application supervisor. If it is found that changes are always assumed to be correct and as a result errors occur, then this is easily highlighted. Consequently, if visual inspections are always performed, especially for 'simple changes' or changes done by experienced personnel, then this suggests overcaution (and expense).

Observations about the QIR

The QIR has been warmly appreciated by all members of development staff. In particular

- Senior members are :
  - comforted by the auditability of changes made by juniors
  - able to confidently delegate more work to juniors
- Junior members are :
  - comforted by the 'checking' mechanisms in areas where they feel insecure about ability
  - appreciative that 'greater responsibility' tasks are delegated to them.

As a result, the number of recorded errors reported by users fell, at the time the QIR was introduced, from 2.3 errors a month to 0.4 errors a month on average, per application. Prior to the introduction of the QIR both logic and consistency errors were found in about equal proportion. After the introduction of the QIR the majority of the errors were process logic errors with consistency errors being almost eradicated.

6.6 Retrieving and Manipulating the Data

In order to analyse the data, a utility program was written which accesses the log file in an 'easy to navigate' fashion. Due to the nature of the data and the fact that the same information is reported from different viewpoints, printing the data on a hardcopy device would be impractical except for one-off purposes. The analysis tool has the following two displays.

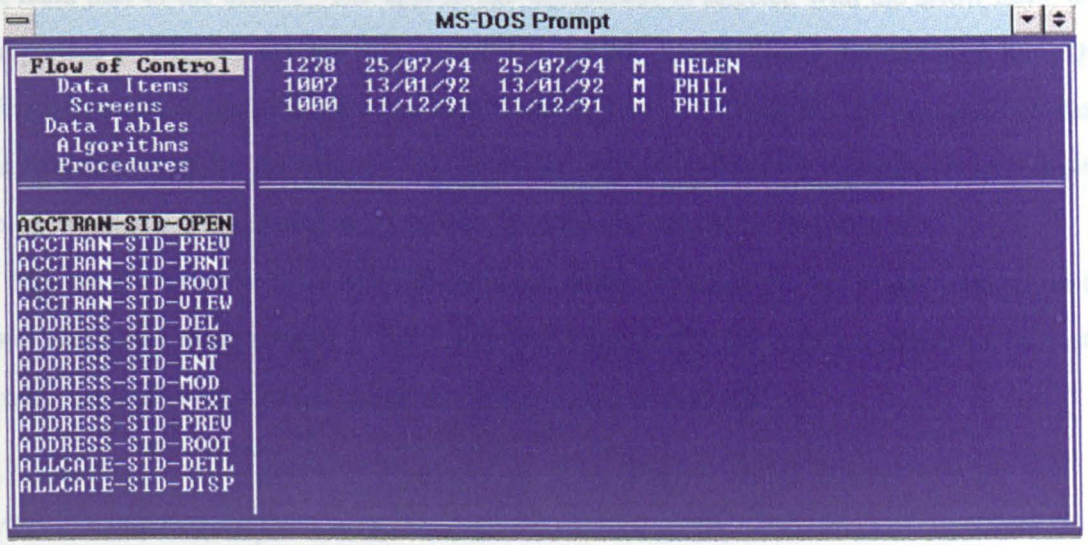

figure 6.7a "Analysis tool : Component Selection"

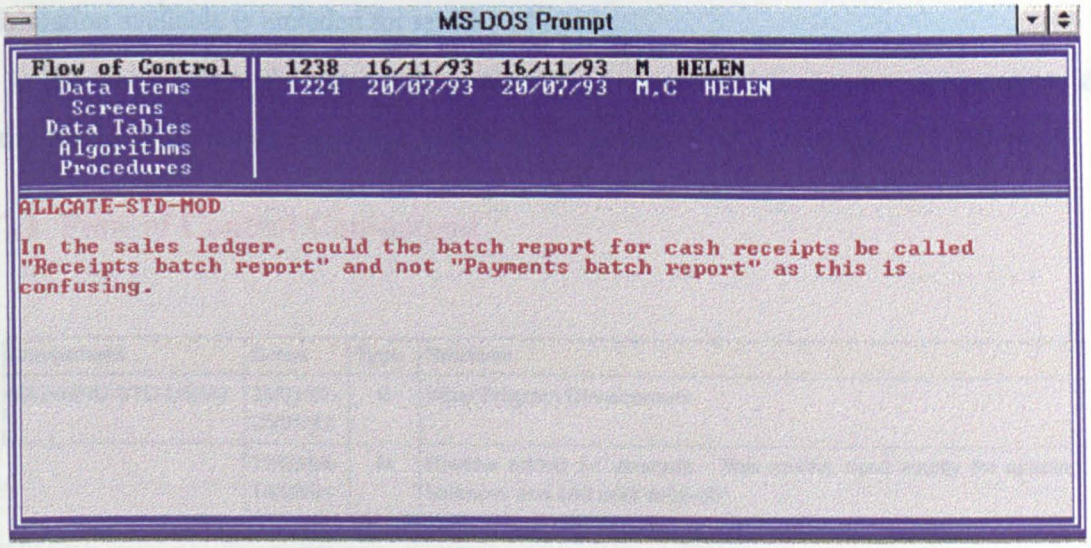

figure 6.7b "Analysis Tool : View Descriptions"

This analysis tool allows the selection of a component type from the top left hand corner as shown in figure 6.7a. A list of all the components of this type are then displayed in the lower

left hand part of the screen. The appropriate component is selected with the arrow keys, as a component is selected the top right hand area of the screen shows the different RPU/QIR references which contains a reference to this component. This contains the start and end dates that entries were made, the names of the development staff making the changes and an indicator which highlights if the entries were created, modified or deleted by this routine.

Moving the cursor to the RPU/QIR reference section expands the "why" descriptions on the lower half of the screen (figure 6.7b). These are dynamically displayed as each entry is located in turn. An option to print the complete design history is available.

6.7 Some Example Data

For the purpose of presenting the results, selected samples from the data will be used. These samples are not chosen because of a 'best' result but to a 'typical' result A history from each of the state components will be included. In the representations below only a selection of the information available is included for reasons of clarity.

In the tables below, the dates refer to the dates between which changes were made to the entity for the given reason. The type means "C" : Created "M" : Modified "D" : Deleted.

6.7.1 Flow of Control Component

Component	Dates	Type	Reasons
MAINMNU-STD-MENU	25/01/92-25/01/92	C	Initial Program Development
	12/03/94-14/03/94	M	Routine added for Journals. This routine used mainly for opening balances and bad debt write-off

The information gathered about these type of components tended to be limited usually giving a single entry for the creation of the routine and perhaps entries where major functionality is added at a later stage. This makes 'sense' as these types of elements simply make menu structures. These are possibly the least important parts of a design.

6.7.2 Data Item Component

Component	Dates	Type	Reasons
INICVAL01	19/06/95 - 19/06/95	C	Add a "Cost Price" field, Sales Order References (Contract No, SO No, Del Note No) and ten user definable fields to the sales invoice header and item files to maintain compatibility with the integrated Sales Order Processing module

The information gathered about these type of components again tended to be fairly limited. By their very nature, data items are not subject to a high degree of change. What is important is that reasons are now "automatically" appearing as to "why" these elements were added. In the example above the reason being for integration purposes.

6.7.3 Screen Component

Component	Dates	Type	Reasons
ACCOUNT	13/01/92-20/07/92	C	Initial Program Development
	26/10/92-26/10/92	M	Make the balance forward field readonly and the turnover field modifiable
	27/10/92-27/10/92	M	When setting a default nominal code, the detail code should be optional, even if a detail code is required for the nominal ledger integration. This allows greater flexibility during system start-up. The batch close routines will check this anyway.
	21/10/93-21/10/93	M	Add a new discount field to the sales ledger account for prompt payment.
	01/12/93-01/12/93	M	Highlight trade and prompt payment discount fields when the sales ledger is not linked to the nominal ledger

The information gathered about screen components begins to build a picture of the data and becomes almost self-documenting. In the above descriptions, only one of the changes (Dec '93) was a 'bug fix'. All the others were enhancements.

6.7.4 Data Table Component

Component	Dates	Type	Reasons
SLEDGER	21/02/92-24/02/92	C	Initial Program Development
	15/12/92-15/12/92	M	Add a new index to the Customer Master file on the Name field. Add a 'find' command to the account screen which does a browse using this index.
	21/10/93-21/10/93	M	Add a new discount field to the sales ledger account for prompt payment.
	09/12/93-09/12/93	M	Add five more user definable fields to the account. These are used as the default when raising sales invoices
	21/08/95-21/08/95	M	Add a flag to the account maintenance page to be accessed by the Sales Order Processing module and to indicate that the customer requires delivery notes to be posted to a single invoice.

Components of a Data Table type have a lot of similarity to Screen type components. In the example above, two of the entries (October 93 and December 93) are identical. They are, in fact, the same RPU. This makes logical sense as in the above example the screen component is based on this particular Data Table with a result that where entries are made to the table they are usually added to the screen. This is not always the case as in the entry August 95, here the new field added to the table is accessed only from a different area.

6.7.5 Algorithm Component

Component	Dates	Type	Reasons
CLOSE_INVOICE	29/01/92-16/07/92	C	Initial Program Development
	26/10/92-26/10/92	M	Closing a Sales Invoice Batch. Even if the stock control posting is set to "N", the sales invoice details should still be posted to stock if the path is set up and the identity exists.
	01/04/93-01/04/93	M	Tidy up batch close routines - if posting to a period other than the current period, update the current balance and turnover but not the invoices or payments this period.
	10/05/93-10/05/93	M	Closing the SL invoice batch; if the Sales Analysis path has been set up the program assumes that the Nominal Ledger path has been set up and the databases opened.
	15/07/93-15/07/93	M	Implement Prompt Payment Discount. Also allow entry of discount amount on the sales invoice items (i.e. override the percentage calculation).
	05/08/93-05/08/93	M	Change posting from invoice batch routine to Sales Analysis. Field in SATRANS have been renamed.

Component	Dates	Type	Reasons
	29/09/93-29/09/93	M	Post Sales Invoice value to Stock Audit Trail. Post cost price to Sales Analysis
	05/10/93-05/10/93	M	Need to post the Product Group to Sales Analysis when closing a sales invoice batch.
	05/10/93-05/10/93	M	Post the Period Number to Sales Analysis when closing a sales invoice batch.
	25/07/94-25/07/94	M	Batch close always updates the turnover figure. It should only be updated if posting to the current year.
	04/08/94-04/08/94	M	The invoice batch close routine should take into account the Summary flags now maintained in the Nominal Ledger for VAT etc.
	14/12/94-14/12/94	M	Allocate transaction numbers from the internal counter in globals instead of adding 1 to the last transaction in the file as this can cause problems in a multiuser scenario.
	07/04/95-07/04/95	M	The Sales Invoice batch close routine does not post a Cost Price if the invoice item does not have stock identity. It should post the standard price from the stock master record.
	12/05/95-12/05/95	M	Post the Sales Invoice date to the stock control record and not the current date.
	19/06/95 - 19/06/95	M	Add a "Cost Price" field, Sales Order References (Contract No, SO No, Del Note No) and ten user definable fields to the sales invoice header and item files to maintain compatibility with the integrated Sales Order Processing module
	11/08/95-11/08/95	M	Add automatic logging of changes to system parameters and batch close routines.
	02/11/95-02/11/95	M	Add the Sales Order Section number to the Sales invoice item file and implement a new Invoice print routine which sorts the items on this field.
	21/12/95-21/12/95	M	When closing batches, validate the period number to stop 'wild' numbers from being entered.
	21/12/95-21/12/95	M	Want the period number to appear on all batch reports

These types of components attract the most useful information from the history log. In the above example all changes made to a central "batch close" routine are listed. Some are bug fixes, others are enhancements. What is highlighted here are peculiarities which may not be easily understood from examining source code directly.

6.7.6 Procedure Component

Component	Dates	Type	Reasons
STATEMENT_CREDIT	15/11/93- 15/11/93	C	Can we have an option in the Sales Ledger which prints statements for all customers with a credit balance.

Procedures by their nature will not attract a lot of modifications. Procedures are simply convenient ways of linking Screens, Data Tables and Algorithms. All the modification tend to be done at this lower level.

6.8 Using the results to transfer 'experience'

The remainder of this chapter describes how the data presented above is used as experience. The aim is to transfer the knowledge which is gained by a developer when developing an application to a developer who is subsequently modifying the application. The reason for transferring this data is to provide the developer with enough information that he/she can modify the program without causing consequential damage.

6.8.1 A Worked Example

This is best explained with an example. The example chosen is the algorithm taken from the sales ledger called VIEW_ACCTRAN. There are two questions that can be asked :

- What is this routine meant to do?
- If it is changed, what are the possible consequences ?

Using the information extracted from the Phase repository, these questions can be answered (albeit not necessarily completely).

The history log attached to this algorithm provides the following information :

Component	Dates	Type	Reasons
VIEW_ACCTRAN	23/06/94-23/06/94	C	Store each invoice printed to a file, referenced by invoice number, and be able to view the invoice from the account transaction browse (SL/1281)
	03/11/94-03/11/94	M	When using the view facility of a invoice, put the screen into condensed mode (SL/1292)
	29/11/94-02/12/94	M	Add the option to print archived invoiced (presently only allowed to view them) (SL/1293)

The first entry in the log explains why the algorithm exists in the first place : To view an invoice which had previously been printed. The second two entries provide further information, the screen is in condensed mode and that there is an option to print.

For further information, it is possible to check which other entities in the repository were affected at the time the algorithm was created or modified.

SL/1281		
Entity Type	**Entity**	**Remark**
Screen	LSETUP	Modified
Algorithm	DISPLAY_SYSPARM	Modified
	MODIFY_SYSPARM	Modified
	PRINT_INVOICE	Modified
	VIEW_ACCTRAN	Modified
Node	ACCTRAN-STD-VIEW	Created
Procedure	ACCTRAN_VIEW	Created
Data Item	INVPATH	Added
Data Table	SLGLOBAL	Modified

This tells us that a data item (called INVPATH) was added to the data table (SLGLOBAL) and is maintained via the algorithms DISPLAY_SYSPARM and MODIFY_SYSPARM using the screen LSETUP. (This table, screen and these algorithms refer to the 'System Control' file containing all the configuration parameters of a module). The data-dictionary remark for this data item tells us that this item refers to a directory path where invoices are to be stored. The only other routine affected is the PRINT_INVOICE algorithm. This is the routine which

creates a copy of the invoice, in the named directory, whenever the invoice is physically printed.

Checking the log for the second alteration provides the following information :

SL/1292		
Entity Type	**Entity**	**Remark**
Algorithm	VIEW_ACCTRAN	Modified

This tells us that no other entity was affected by changing the screen mode. This third change is listed below.

SL/1293		
Entity Type	**Entity**	**Remark**
Algorithm	PRINT_ACCTRAN	Modified
	VIEW_ACCTRAN	Modified
Node	ACCTRAN-STD-PRNT	Created
Procedure	ACCTRAN_PRINT	Created

This tells us that the 'Print' option is actually contained in a separate procedure with a separate algorithm. The VIEW_ACCTRAN macro does not actually print the invoice but must set a pointer to the invoice viewed which is used by the print routine.

From this information we now have an understanding of the functionality of this routine, and we also know that changing this routine will not affect any other part of the program. This prepares the developer for making any changes to the routine and will provide clarity when reading and amending the algorithm code.

6.9 Conclusion

This chapter has described an implementation strategy for collecting information relating to changes made to a program developed using one of the Phase CASE tools. The information collected is simple : what has changed, and why has it changed.

An analysis tool has been designed for use by developers who are familiar with both the application domain and the Phase paradigm. The information presented represents the history of the development of a program, or components of a program and contains many of the decisions made by previous developers. This information is extremely useful when modifying or maintaining parts of a program, both in helping with an understanding of the construction of an application and also the implications and consequences of change.

This chapter and the previous chapter have described the Phase paradigm, the functionality of the CASE tools required to develop applications using this technique and examples of how the information collected by the system can be passed on and used by developers. The next chapter provides examples of studies based upon the use of the Phase paradigm in a commercial environment. This provides an indication of how resilient applications developed in this way are to the detrimental effects of changing mature programs.

Chapter 7

The Phase Resistance to Change?

7.1 Introduction : How does a Phase program perform ?

The purpose of this chapter is to provide an answer to the question:

"How does the Phase paradigm score, in relation to the consequence of changes, when applications which are developed using the technique are subject to change?".

To provide a tangible measure of success, two key measures have been monitored. These are :

- Time taken to complete the change, measured in Programmer Hours (PH)
- Number of reported errors in the system after the time of 'release'

The experiments used in this analysis are not laboratory exercises but 'real' examples taken from applications developed in a commercial environment. The advantage of using these real examples is that they reflect problems of a significant complexity and funding was available to provide solutions to these problems. The disadvantage is that it is difficult to provide a direct comparison of the consequence of change of a single system developed using different methodologies.

Five examples are included in this chapter, three concerned with technological changes and two concerned with major changes in user requirements. The conclusion to this chapter

includes an analysis of the contribution of the various specific elements of the Phase paradigm to its overall success in resilience to the effects of change.

The examples are :

- Technological Change
  - Change of target language to the 'next generation' of the compiler
  - Change of target language to a different platform
  - Change from a procedural language to an event driven language
- User Requirements Change
  - Adding significant enhancements to a mature software program
  - Maintenance of software by non-original team members

7.2 Change of target language to the 'next generation' of the compiler

The software produced by EDS used a target language compiled by a 1987 version of a compiler. This compiler was replaced by the 'owners' and became unsupported in 1992. Dispite this, due to the effort of the development team on the functionality of Elite programs, this target language was still used up to the first quarter of 1995. It was the limitations of poor memory management intrinsic to this technology which placed a practical limit on the ever increasing functionality of the programs.

The successor to this target language compiler was not classed as 'backward compatible'. It had a basic similarity in syntax to its predecessor however it had significant differences in the way that programs could be structured in terms of information hiding capabilities. For comparison, the differences were similar to the relationship between a Pascal program and a Modula-2 program.

In 1995, the first real test was applied to the Phase Paradigm, to 'upgrade' all Elite modules to the new compiler with the minimum of effort. A 1 month exercise was scheduled to 'learn' about the new compiler and highlight the differences. This produced the following results.

- The existing syntax of all application code definitions could be regenerated to the new target language by programming direct translation 'rules' into the code generator with few exceptions.
- Manual changes to application code were minimal and due completely to 'bad programming style' of the algorithm definitions
- A number of 'third party program libraries' had to be replaced and the functionality of these routines recoded either 'by hand' or by finding alternative libraries
- The Kernel routine required only minimal changes to take advantage of improved memory configurations.
- No changes were required to the repository specification

After a period of initial testing, each module was moved to the new compiler. The following table provides statistics for the 10 major modules.

Module	Complexity Factor	Time for Completion	Errors Reported
SL : Sales Ledger	1	9 PH	1
PL : Purchase Ledger	1	7.5 PH	2
NL : Nominal Ledger	1.3	6.5 PH	0
PY : Payroll	1.7	10.2 PH	1
SO : Sales Order Processing	1.1	8.5 PH	2
PO : Purchase Order Processing	1.4	13.5 PH	0
ST : Stock Control	2.3	7.5 PH	1
SA : Sales Analysis	0.4	9.5 PH	0
PA : Purchase Analysis	0.3	7.5 PH	0
JC : Job Costing	2.1	12 PH	1

Complexity Factor

The **Complexity Factor** in the above table is a metric calculated to provide a guide for comparing modules in terms of complexity, using the reference of the Sales Ledger, having a

complexity of 1. This complexity factor is a calculation based on the number and type of algorithm entries in the repository. Algorithm entries can be split into two types :

- Type A1. These relate to the standard Display, Enter, Modify, Delete, Browse procedures as detailed in section 5.3.2
- Type A2. These relate to the non-standard routines eg Batch Close, also described in section 5.3.2.

Monitoring the effort taken to implement, test and perform maintenance on procedures using these algorithms over a period of a year provides a rule of thumb which estimates the effort required for a procedure with a type A2 algorithm to be 5 times greater than the effort required for a procedure with a type A1 algorithm.

The complexity factor for a module is calculated by the formula :

$$\frac{\text{number of procedures using type } \mathbf{A1} \text{ algorithms} + (5 \times \text{the number of procedures using type } \mathbf{A2} \text{ algorithms})}{\text{base factor for the sales ledger}}$$

The base factor for the Sales Ledger is (using the 'size' factors given in Chapter 6) :

$$103 + (5 * 31) = 258$$

Example

The complexity factor for the Nominal Ledger using A1= 120 and A2 = 43 is :

$$\frac{120 + (5 * 42)}{258}$$

This equates to 1.3.

This figure does not have any real scientific meaning, however, it does provide a means of comparing modules and provides an indication that the Nominal Ledger is about 30% more complex than the Sales Ledger. The results produced by this simplistic formula are similar to intuitive factors placed on the complexity of the above modules by programmers who are familiar with the modules.

Time Taken for Completion

The time taken for completion was measured in Programmer Hours. These were 'clock hours' recorded on manual timesheets to include the following :

- Execute the automatic translation routines
- Manually correct any syntax errors reported by the new compiler
- Recode routines which relied on 'old compiler technology' (regarded as bad practice)
- Relink all object files
- Execute a test set of data for all functions
- Execute each non-standard algorithm using 'real' data

Figure 7.1 plots the relationship between complexity and programmer hours.

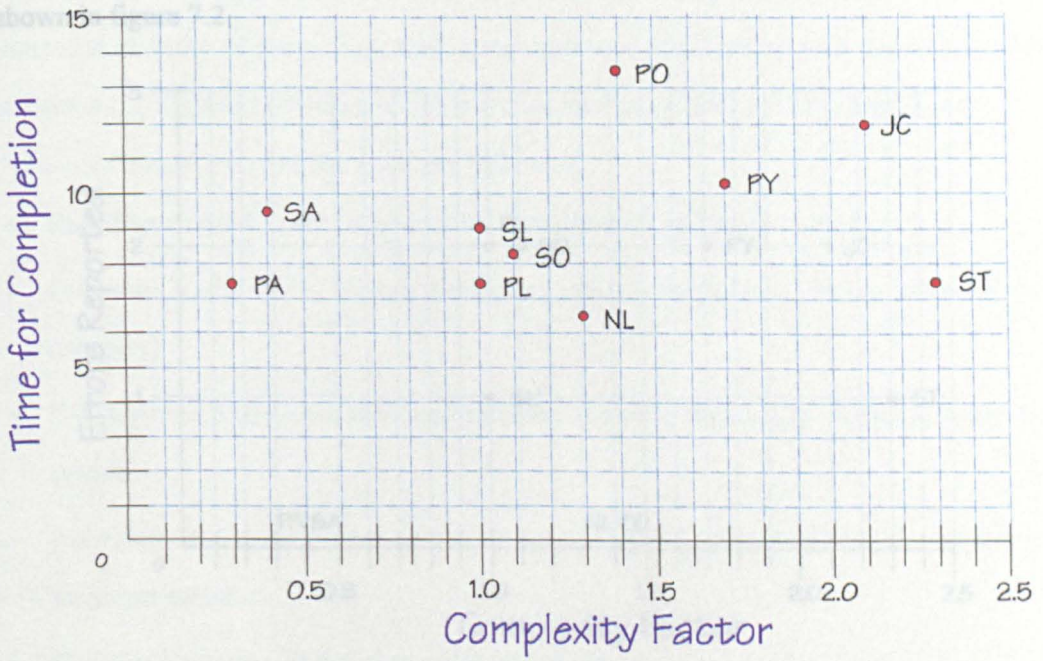

figure 7.1 : Relationship between complexity factor and time for completion

It can be seen that there was a minimum period of 7.5 programmer hours per module. This represented the time taken mainly to execute the test set of data for all functions.

There is no direct relationship suggested between the complexity of a module and the time taken to implement the module in the new target language.

Errors Reported

The number of errors reported referred to errors which caused the program to terminate abnormally after the program was released to users. These are errors which 'slipped through' the test data and test procedure. All the errors reported were due to 'bad programming style' where programmers had not followed the standard code of practice when coding unstandard algorithms. The relationship between the complexity factor and the number of errors reported is shown in figure 7.2.

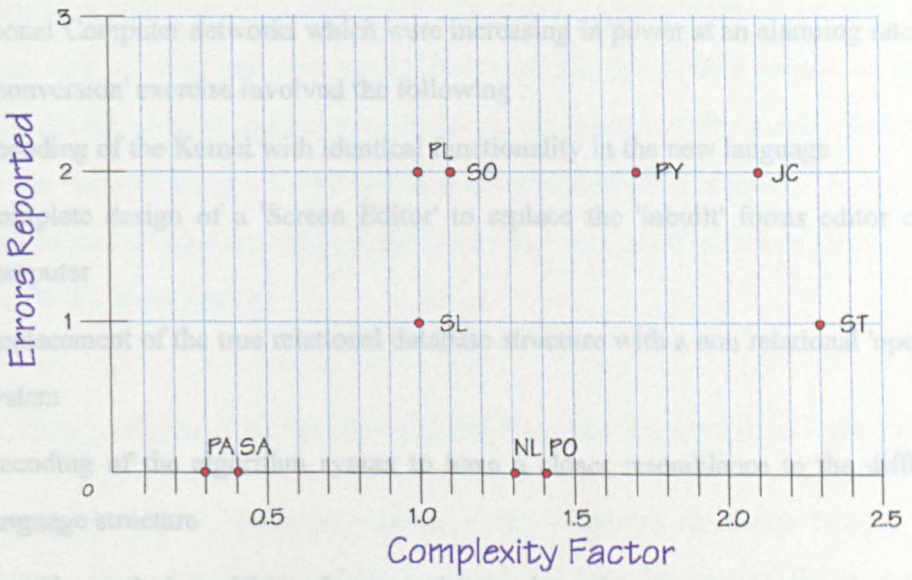

figure 7.2 : Relationship between complexity factor and number of errors reported

No direct relationship was found between the number of errors reported and the complexity of the module.

Summary

From the data given in the above experiments it would suggest that changing technology to a compiler of a similar structure is relatively easy to perform for programs written using the Phase paradigm.

7.3 Change of target language to a different platform

The only time that software developed using the Phase paradigm has been changed to a different platform was when FDS, a Pascal system on a mini computer, was rewritten as EDS, a Clipper system on PC networks in 1990.

Unlike the previous exercise, the change in technology in this example was forced due to a redundancy of the support of hardware and associated Operating System platform. The commercial viability of proprietary mini computers was diminishing with the advent of the low cost Personal Computer networks which were increasing in power at an alarming rate.

The 'conversion' exercise involved the following :

- Recoding of the Kernel with identical functionality in the new language
- Complete design of a 'Screen Editor' to replace the 'inbuilt' forms editor of the mini computer
- Replacement of the true relational database structure with a non relational 'open' database system
- Recoding of the algorithm syntax to have a closer resemblance to the different target language structure
- Complete redesign of the code generation routines

No 'computer performed' translations were attempted as even basic file transfer between the platforms was impractical, all the functions of EDS were coded 'by hand'.

The Timetable

The complete conversion of FDS to EDS can be divided into two parts. FDS (and EDS) are both programs which are 'written in themselves'. They can be considered as simply an

application (with the functionality and purpose of a CASE tool) which is implemented using the appropriate Kernel and Support Library.

To recode the Kernel and Support Library took approximately 210 Programmer Hours (over a 3 month period). Much of this time however was experimental and included the learning curve of a new language. It also included the creation of a screen editor as the intrinsic 'forms' editor available with the mini computer was not available with the PC network.

Developing the application (EDS) using the new support library and Kernel took approximately 120 Programmer Hours (over a 2 month period). This figure eventually rose to 430 Programmer Hours (over a 4 year period) as a number of enhancements to the system were added.

Summary

The example above is indicative of a 'worst case' scenario, where no automatic re-use of information is possible due to technological differences. The examples above also involved an application with a very high complexity factor of 9.2 (as defined earlier in this chapter) compared with the 'normal' range of complexity factors found with the previous examples. This is due to the example being a CASE tool and not 'standard' IBIS software. In my opinion the time taken to perform this transition is intuitively low compared with time taken in the past to migrate software 'from scratch'.

7.4 Change from a procedural language to an event driven language

During the early 1990's, Microsoft introduced the Windows Operating System for the PC environment. This provided a platform for Graphical User Interfaces (GUI) and event-driven programming. Dispite an initial reluctance, the commercial market became dominated with "Windows Software" around 1995, the use of MSDOS programs diminishing.

Programming with the GUI environment had two significant problems :

- The use of 'forms' and screens had to conform to a defacto Windows standard
- Programs had an 'event driven' look and feel utilising the concurrent nature of the operating system

Replacement of screens to 'Windows Standards'

A three month experiment was performed in the summer of 1996. The purpose of this experiment was to replace the standard user interface of Elite programs to the defacto (Microsoft) Windows standard. A constraint was added that no application code should be altered.

This was possible due to the use of the Support Library for accessing the User Interface. From an application code view point single function calls placed 'screen definitions' onto the physical VDU. These were either 'detail screens' or 'browse screens' (as described in chapter 5) . Menus and command options were part of the 'Kernel' routine and already separate from the application code. The screen editor in EDS was modified to handle the graphics capabilities of windows screens.

The experiment concluded that a transformation to a GUI was possible without changing any application code and restricting all changes to the Kernel and the Support Library. The level to which users accept the system as 'Windows Software' however has not been tested at this time.

Replacement of Flow of Control from Procedural to Event Driven

No attempt has been made to alter applications to a true event driven environment due mainly to the large cost implications. There is a lack of a 'ready made' suitable development tool and the resources available for research and experimentation in the commercial organisation are limited. It is predicted however that changing to an event driven paradigm would not be a significant problem (at least no more significant that the conversion from FDS to EDS). The remainder of this section describes how it would be attempted.

In Phase terminology, a collection of related flow of control nodes is called a SubModule. For nodes to be a SubModule they must be a hierarchically related subtree in the main flow of control structure and all perform 'actions' on a common data table or screen. For example, the nodes to Display, Enter, Modify and Delete records in the customer file are a SubModule. This has a very close relationship to an Event Driven (or Object Oriented) paradigm. A Submodule would become an OO Object and each node would become an OO method acting on the object.

An Example

This can be clarified with the aid of an example (simplified) :

In Phase assume we have the table DBACCOUNT with the following structure :

Field	Description
AC_CODE	Account Code
AC_NAME	Account Name
AC_BALANCE	Current Balance
AC_TURNOVER	Value of Sales Invoices in year ex VAT
AC_VAT_NO	Account VAT number

In Phase, manipulation of data in this table would be represented by a flow of control structure :

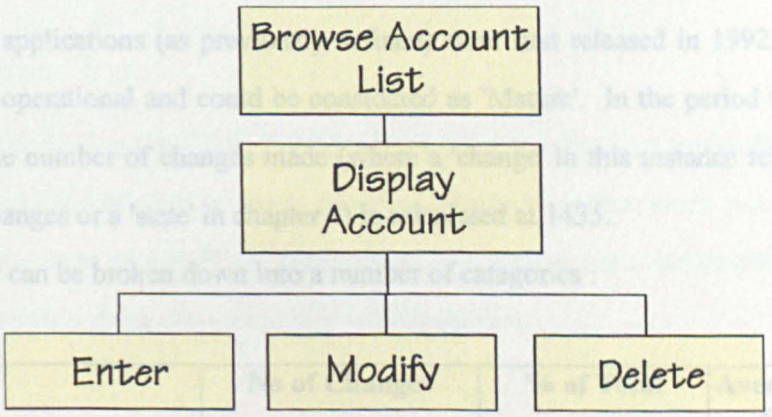

figure 7.3 : Standard Phase SubModule

In Object Oriented terminology, this would be replaced with the Object Definition :

```
Define Object Account;
    export methods Browse_Account ,
                   Display_Account,
                   Enter_Account,
                   Modify_Account,
                   Delete_Account,
    export structure Ac_code,
                     Ac_name,
                     Ac_balance,
                     Ac_turnover,
                     Ac_VAT_Number
```

All the information required to automatically generate these Object definitions is held within the Phase repository structure.

Summary

As described above, there is a high degree of similarity between the structure described by the Phase paradigm and the structure inherent in the Object Oriented paradigm. The discussions above indicate that implementing a Phase program in an event driven manner would be relatively straight forward.

7.5 Adding significant enhancements to a mature software program

The 'core' Elite applications (as previously defined) were first released in 1992. At this point they were fully operational and could be considered as 'Mature'. In the period from July 1992 to July 1996, the number of changes made (where a 'change' in this instance relates to a set of related entity changes or a 'state' in chapter 4) is calculated at 1435.

This number can be broken down into a number of categories :

Category	No of Changes	% of Total	Average Time to Complete
Major Enhancements	367	25.6%	15.75 PH
Minor Enhancements	612	42.6%	4 PH
Cosmetic Enhancements	272	19.0%	0.5 PH
'Bugs' as a result of the original development	147	10.2%	2.5 PH
'Bugs' introduced during maturity	37	2.6%	0.75 PH

- Major Enhancements relate to user requests which provide additional 'goals' for the software over and above the requirements of the software at the time of initial release. This is similar to the second case study example given in chapter 4. Typically, this involved adding or changing more than 20 entities in the repository.

- Minor Enhancements relate to user requests which can be considered as 'refined' requirements. The number of entities changed in the repository is between 5 and 20.
- Cosmetic Enhancements relate to user requests that do not change the functionality of the system, but simply the layout of screens or reports. The number of entities changed in the repository is less than 10.
- 'Bugs' as a result of initial development relate to errors found in the software which were present (although undetected) in the system at the time of release. The number of entities changed in the repository is usually low.
- 'Bugs' introduced during maturity relate to errors found in the software which were introduced after the initial release, usually as a consequence of introducing enhancements.

Summary

Programs developed using the Phase paradigm have been significantly modified after the initial release. Intuitively the number of errors introduced into the system seem low compared with historical developments of non-Phase applications of a similar (or smaller) size. The average time taken to complete these changes is also intuitively low.

7.6 Maintenance of software by non-original team members

Of the 1435 changes made to the programs presented above, 72% of the changes were made by programmers who were not involved in the original development of the modules. This was due both to staff turnover and the reorganisation of the development team structure. As a general point, it was a policy decision to 'move developers' between modules after a period of 6 months. This prevented any one individual from having 'ownership' of a piece of software which, from previous experience, ensured that any 'bad' programming style became apparent.

Summary

Programs developed using the Phase paradigm have been significantly maintained by programmers who were not involved with the original development. The statistics have been described in section 7.5 above and it has been shown that the number of errors introduced has not been significant.

7.7 Dynamic Configuration at RunTime

During 1994, the advantage of the Support Library for all access to the repository was utilised to its fullest potential. It was realised that as high a level as 80% of a users specific requirements was based around 'jargon' and 'terminology'. As an example, in simple terms, the functions of a commercial business can be summarised as follows :

- Generate Sales Leads from Customers
- Log a Sales Order
- Place Purchase Orders on Suppliers for 'Raw Materials' or Services
- Provide a product or service to your customer
- Send an Invoice & Receive Payment
- Provide Support

Whilst the general functionality is similar, the terminology and specific requirements are not. Customer Orders may be called : Customer Orders, Sales Orders or Jobs. Orders may be stock related, made to order or manufactured.

This led to the observation that whilst the functionality of the algorithms provided a 'standard set of functions' and rarely required alteration; menu labels, screens and reports required a high level of alteration to suite particular users. As previously discussed, all these aspects of the User Interface were driven though single calls to the Support Library.

By altering the Support Library routines and providing a 'Run Time' subset of the repository containing the Screen, Browse and Menu Option Labels it became possible to alter these entities for particular installations without changing any of the original repository entities. By

creating a run-time definition only for entities which were actually changed (or customised), program updates could easily be accommodated.

Summary

A large number of configurations of a single Phase module can be accommodated using data driven techniques without altering the base programs. This has the effect of removing a significant load from the development teams as well as reducing the errors introduced by the development activity.

7.8 Conclusion : Why is the Phase method a good method ?

From the above exercises, it can be concluded that Phase programs can be modified and changed without extreme difficulty. The remainder of this chapter provides a closer examination of the following question :

Why is a Phase program reasonably resilient to the detrimental effects of change ?

A focus group of developers using the Phase method was formed in order to establish a set of issues which contributed to the success of the Phase method in being resilient to the detrimental effects of change. The issues raised are listed below and presented in order of decreasing importance.

7.8.1 Run Time configuration of the User Interface

Configuring the User Interface at Run Time and external to a program, to reflect specific user terminology and requirements removes a significant source of complexity from a program. A major source of change is when software has to satisfy the needs of multiple users, each with their own particular requirements. Using data-driven techniques each user can have a different configuration of the same program.

7.8.2 High Coherence of Procedures

Procedures in a Phase program must be completely independent of any other procedure. The only parameter passing mechanism is a 'current' entry in a data table. Changes to procedures can be made in total isolation from any other procedure.

7.8.3 Use of Prototype Software

It has been observed that in order to achieve a reasonably stable requirement, a user needs approximately 3 iterations of refinement. If prototypes are not done, these iterations are (expensive) development exercises. Prototypes are very inexpensive to achieve using the Phase paradigm and can be 'thrown away'.

7.8.4 Ability to view the 'history' of an entity within the repository

The advantage of knowing why entities have a particular set of attributes is significantly more beneficial than simply knowing what the attributes are. This transfers vital knowledge between developers and is used to provide better information, allowing more informed design decisions to be made during maintenance. It also aids the development of the conceptual model as described earlier in this document.

7.8.5 Printing and checking of the Quality Inspection Record

The QIR enforces discipline on developers to both complete their task and check that what they have changed within the scope of the requirement for the change.

7.8.6 Easy availability of documentation

Easy access to documentation provides vital information to developers quickly. This accurate information reduces the 'guesswork' and subsequent errors.

7.8.7 Mental Model of an application represented by the Flow of Control Tree Structure

The ability to present the 'mental model' of a system to developers provides a high 'comfort' factor when trying to understand the functionality of a new module or program.

7.8.8 Use of automatic code generation for repetitive tasks

Automatic generation of code guarantees consistent programs. This also leaves much more time for programmers to concentrate on the more complex and non-autogenerated functions.

7.8.9 Automatic Hyperlinking of Entities

The automatic hyperlinking of entities within the repository provides an automatic method of knowing which entities will be affected by changes

7.9 Summary

This chapter has shown that programs developed using the Phase paradigm are more resilient to the detrimental effects of change than similar programs developed historically using more traditional methods by the example software development company. The main contributing factors have also been discussed.

Chapter 8

An Assessment of Phase

8.1 Introduction

This chapter examines the Phase paradigm in relation to its original goals and objectives. For analysis purposes, the Phase paradigm can be assessed for its effectiveness as

- A requirements analysis tool
- A specification representation system
- A software designers productivity tool
- A software project management system

The ultimate goal is the resilience of a Phase project to the impact of changing requirements. This chapter includes data, statistics and observations taken from Phase projects over a five year period and assesses its effectiveness in relation to change. The attitude of software developers towards change has been related directly to the tools which are available to them. The success of the Phase paradigm in achieving its ultimate 'resilience to change' goal therefore has a direct correlation with the attitude of the developers who use it.

Each section in this chapter is introduced with the source of the evaluation criteria which is taken from literature where appropriate.

8.2 Phase as a Requirements Analysis Tool

Phases uses a prototyping approach as an aid to requirements analysis. Objectives of a prototyping mechanism are given in [Floyd83] and it is these objectives that will be used as a 'benchmark' for comparing the effectiveness of the Phase prototyping properties. In her analysis of 'ideal' prototyping features, Floyd proposes the following major headings:

- Early Availability
- Demonstratable/Executable
- Construction
- Commitment to Target System
- Documentation
- Automated Program Generation
- Further Use

8.2.1 Early Availability

To allow maximum effectiveness, a prototype must be available as soon as possible after the initial systems analysis has taken place. This provides maximum enthusiasm for a system as users are provided with feedback at an early time. This helps avoid the problems of users 'forgetting what they have said' which happens in the time between the initial analysis phase and their feedback.

In the Phase system, due to its construction (see below) a prototype is easy to prepare. A complete prototype can be created for a 'typical' project in a single day. At most, users involved in a Phase development will expect to get feedback within 1 week.

8.2.2 Demonstratable/Executable

The prototype review is a critical part of any prototyping strategy. For maximum effectiveness the prototype should execute in such a way that users can get a true 'feel' for the software before it is developed.

In the Phase system the prototype specification is executed by interpreting the high level prototyping instructions. These instructions will re-create menus, options, screens and prompts in an identical manner to the target program. Delay loops can be incorporated to give the system a 'working feel'.

8.2.3 Construction

The construction of a Phase prototype is described in Chapter 4. To summarise, the structure of the software is created by using a simple tool to create a data-driven directed graph. This graph corresponds to menu options and command options. A simple prototype specification language which consists of only four commands links the structure to the screen and report entities in the specification. The screen painter and report defining tools complete the construction of the prototype.

A Phase prototype is therefore very easy to construct.

8.2.4 Commitment to Target System

A prototype should be committed to its target system. This means that there should be a very close relationship between the prototype (or execution of the prototype) and the execution of the target system. This applies both to the look and executable 'feel' of the software.

The construction of the Phase prototype is such that the entities of the user interface are stored in the repository. These entities that are used during the execution of the prototype for analysis and are also used for the automatic creation of the program, either directly (as in code generation) or indirectly (used during the run-time execution of the program)

The prototype in the Phase system is very committed to the target system. This however leads to a major disadvantage which is discussed later in this chapter.

8.2.5 Documentation

As well as an executable prototype some traditional forms of documentation are also desirable. This includes flow diagrams, data structure diagrams, reference and technical manuals.

The Phase system has a number of automatic documentation tools which have been described in chapter 4. These are summarised as :

- **Flow of Control Node Tree Structure Diagram**. An automatically produced pictorial representation of the structure of the menus and options within a software product. This can be printed at various levels of abstraction, from a single A4 sheet where each 'node' is simply numbered to the level where each node automatically prints out the corresponding screens. In the later case, the full diagram for a typical module, showing all screen activity at every node can exist as a 14 metre by 3 metre chart. (Automatically split into A4 pages).
- **Database Definition Charts**. Standard reference manuals detailing database structures, indexes and relationships.
- **Screen Definition Technical Reference**. A programmers reference for the information and structure of screens defined using the screen painter.
- **Module Technical Reference Manual**. This is a hardcopy document which displays screens and options combined with notes in a 'user oriented language' which can be used for reference. This information is also available as an on-line context sensitive help within the run-time versions of the finished products.
- **User Tutorial Manuals**. These are 'how to do it' guides which contain text and links to appropriate screens.
- **Hard Copy Prototype Manual**. This is a 'flat' hyperlink structure document which is highly cross-referenced automatically between pages. This is ideal for recording notes on the prototype as it is being executed to users.

The major advantage to all of these documentation forms is the close integration to the repository. Where elements are changed in the repository to alter functionality of a program, the 'documentation' can be reprinted without further amendment.

8.2.6 Automated Program Generation

Automated program generation is desirable as a feature which should be available in all prototyping systems. The prototype is a specification of the system which, if it has the

properties of commitment to target and is sound in structure, should have the capability of using this information as an input into a code generator.

The Phase system uses automatic code generation at two levels. The first uses the information contained within each screen definition to create fast efficient target code for all screen manipulation. The second uses the information contained in the algorithmic code specification to combine with the other entities in the repository to produce software in its target language. The definitions of these translations are maintained within the system. This allows different target languages to be used from the same specification.

8.2.7 Further Use

The information contained within a prototype specification system should have further use. This has been implied in many of the previous headings in this chapter. Many of the entities are reused as part of the automatic code generation, others are used at run-time, accessed within the program libraries for activities such as flow of control, security etc.

The only element in a specification which does not have some re-use is the small prototyping instruction set which is used solely during the execution of the prototype and automatic preparation of the documentation.

8.2.8 Summary

When compared with the objectives of Floyd it has been shown that the Phase paradigm is effective as a requirements analysis tool. Whilst this thesis is not concerned directly with the analysis methods, what has been provided is a mechanism for rapid prototyping which generically is a recognised method for helping with analysis. It also provides a mechanism for recording the results of the analysis in a structured and reusable manner.

8.3 Phase as a Specification Representation System

Although there are no standard evaluation criteria for a specification, several guidelines, introduced in the 1970's identify some important criteria for the effectiveness of any specification representation scheme. Parnas [Parnas72] stated that the specification scheme

must provide to both the intended user and the implementer all the information needed by each, but nothing more (i.e. information hiding). He also asserted that the specification must be formal so that it can conceivably be machine tested, and it must 'discuss the program in the terms used by the user and implementer alike' (i.e. not some abstract formalism). Liskov and Zilles extended the criteria as follows [Liskov75].

- Formality : It should be written in a notation that is mathematically sound. This is a mandatory criterion.
- Constructability : One should be able to construct specifications without excess difficulty
- Comprehensibility : A trained person should be able to reconstruct the concept by reading the specification.
- Minimality : The specification should contain the interesting properties of the concept and nothing more.
- Wide range of applicability : The technique should be able to describe a large number of concept classes easily.
- Scaleability : The technique should be applicable to applications regardless of size.

The Phase system will be assessed in relation to these criteria. In addition, it will be demonstrated how the Phase system is scaleable and can be applied to applications of a significant size.

8.3.1 Formality

The Phase method is not considered as a formal method [Liskov75] and therefore it is inappropriate to judge it on the basis of being mathematically sound. It has been tested and demonstrated to produce reliable and consistent programs and therefore satisfies the underlying principle that Liskov and Zilles were trying to achieve - that it can produce sound programs.

8.3.2 Constructability

Constructability is the ability to construct specifications without excess difficulty. There are two ways in which this can be measured.

- The time taken to specify a concept

- The number of times a concept specification is changed.

Time Taken

The time taken to specify a concept is influenced by a number of factors including

- The 'size' or 'scope' of the concept
- The complexity of the concept
- The skill of the system designers

It is therefore very difficult to have a suitable single metric for measurement. In general however observations and studies have been made regarding the times taken to construct specifications. A typical 'concept' with a four to six data table scope takes between 6 - 10 hours to develop. This is not dissimilar to non-Phase specifications however it has been observed that Phase specifications can be created by less-experienced junior personnel that can otherwise be expected.

Number of Specification Changes

The number of times a specification is changed in this section relates to the changes required to specify a 'stable' concept, not changes which are due to technological or user requirement changes. The number of times a specification in a Phase system is changed is very easy to identify due to the entity logging functions contained within the Phase editors.

Statistics have been taken and an average number of changes computed for each type of entity within the Phase repository. These are listed in the table below.

Entity Type	Average number of Changes Made
Node	1.3
Procedure	0.2
Screen	3.4
Data Item	0.1
Data Table	1.3
Algorithm	4.1

The 'simple' entities (Procedures and Data Items) are very rarely altered after their initial definition. Screens and Algorithms are edited on average between 3 and 4 times after initial definition, these entities are more complex. Upon further investigation, screen entities are changed mainly for cosmetic reasons, algorithms for further refinement.

In general the number of 'attempts' required to specify a requirement concept correctly is minimal.

8.3.3 Comprehensibility

Comprehensibility requires that a trained person be able to reconstruct the concept by 'reading' the specification. In Phase, the conceptual model in the repository is viewed as a statement of the problem and its solution, it should be 'close' to the designers mental models. Thus Comprehensibility can be demonstrated if it can be shown that a designer who understands the problems being addressed (i.e. domain knowledge) and the specification language (i.e. the Phase system) has little difficulty in creating and revising specifications.

A measure of comprehensibility can be found in the relative ease that designers have in modifying programs that they did not create initially. This can be monitored easily in the Phase system due to the automatic logging of entity changes.

For a 'typical' program a study was made relating the number of changes made to entities after they had first been released. It was observed that over 75% of all entities had changed in the five year period after the first release. Of these over 62% were changed by a developer who was not involved with the initial development. Some algorithms had changed up to 14 different times over the same period by up to 6 different developers. It was also possible to identify the number of changes made for any one requirements change, the average number of changes for algorithms (the significant entity) still remained at an average of 3.2 changes.

This data shows that the Phase requirements specifications are comprehensible.

8.3.4 Minimality

Liskov and Zilles defined minimality in the sense that Parnas defined information hiding. The specification should provide only the necessary information and nothing more. In the 1970's the concepts of interest were data abstractions. In the Phase environment the concepts to be

specified are computer supported responses to a need. Nevertheless, the goal remains the same : to have the implementer specify the interesting properties of the concept and nothing more. Phase addresses this challenge by providing:

- a holistic repository in which all concepts are shared within an application
- a specification command language with very few but very powerful commands
- loosely coupled but highly cohesive 'submodules' of information
- a system style that augments the specification of the interesting properties with the implicit and default properties expected in the implementation.

8.3.5 Wide Range of Applicability

One desirable property of a representation scheme is that it can apply to a wide range of concepts. Phase has been optimised for interactive information systems, but there is nothing in its architecture that restricts it to this class of application. The basic design recognises that requirements engineering is domain oriented, and no one environment (or specification language) will suffice for all domains. Phase uses application-class knowledge to tailor the system to a particular class of needs, and the architecture permits the introduction of system styles for new domains (e.g. real-time and embedded systems).

Even though Phase may be employed with multiple application classes, it has only been primarily demonstrated in the domain of IBIS software. This domain, however, is itself quite broad; it includes standard Order Processing type applications; Automatic Data Collection applications (Shop floor time recording, Aluminium Can recycling scale interfacing).

In addition to IBIS software both the Phase CASE tools (FDS and EDS) have been developed successfully using the Phase method. The application domain of a CASE tool is significantly different from the application domains associated with IBIS software. The Phase CASE tools are more 'database oriented' however than the more traditional concept of a CASE tool which tends to be highly graphical [Junk88].

8.3.6 Scaleability

To demonstrate the scale of an application that can be designed using Phase, the largest Phase project was analysed. This project is the 'core' of the business information modules which are implemented in over 30 sites. The core modules cover all the standard business applications including accounts, payroll, order processing, stock control, costing, and estimating. This set of modules (at 1996) includes over 150 data tables, 370 screens and 750,000 lines of algorithm specification code. At the start of the five year period being studied, the size of core package size was 54 data tables, 140 screens and 321,000 lines of algorithm code. Therefore over the five year period the size of the project has increased almost three fold. This project was maintained by a team of 4 developers who were also responsible for maintaining, installing and supporting a number of other projects. During this time, the system was considered virtually error free (the actual number of errors reported from users over the five year period was 37, an average of 1 reported every 2 months).

This data demonstrates that a Phase project can be significant in size.

8.3.7 Summary

When compared to the guidelines given, it has been shown that the Phase paradigm is effective as a specification representation system.

8.4 Phase as a Software Designers Productivity Tool

There have been many attempts at defining the requirements of the 'perfect' designers productivity tool. The objectives used in this analysis are based upon the observations of Davies and Castell [Davies 92]. They observed that designers follows a similar behaviour pattern :

- Developers create a **mental model** of their design, however, there is great difficulty in representing this model in tangible forms.
- Designers use **mental execution** of the design model as a technique for refining and clarifying the design.

- Designers use **opportunistic development** which includes a mixture of top down and bottom up approaches. Although an overall strategy may be adopted, different techniques will be applied depending upon the particular concept being designed.
- Designers use extensive **note-making** which allow them to records ideas as they happen which may be associated with a different issue in the concept. These notes can be revisited and refined at a later stage.

8.4.1 Mental Model

The mental model used by designers has the definite advantage that it is not constrained by any syntactical issues. It is intuitive to the designer and it can be at different levels of abstraction. Representing this mental model in a physical form requires syntactical constraints.

Experience in designing Phase projects does not preclude mental models, however the mental model is usually created along the intuitive flow of control structure of Phase projects. Physically creating a flow of control structure within the Phase repository allows the "Flow of Control Tree Diagram" to be created. This diagram has a strong similarity to the mental model. This has been proved by 'team driven' projects where the flow of control structure has been created by an individual member, the structure chart printed and distributed to other team members. Observing a team project meeting where the only physical information is this chart, it is soon obvious that each team member creates his own mental model which, from the discussions that follow, the mental models are similar. This proves a strong link between mental models and Phase flow of control charts.

8.4.2 Mental Execution

Mental execution of the mental model is an important part of the modelling process. Experienced designers will be able to execute the mental model derived from the Phase flow of control charts and use this execution to refine and define a design.

The Phase prototyping functions link together elements of the repository which represent the design in a way that can be executed. This physical execution of the prototype can often be a

concrete representation of the mental execution of the model. This shows a strong link between mental execution of models and execution of Phase prototypes.

8.4.3 Opportunistic Development

Opportunistic development involves concurrently using different approaches during design. For example, high level structures may be created across a module, followed immediately by a detailed design of a particular screen. The process of defining the screen may 'spark' higher level thoughts over the methods of obtaining the data associated with the screen and in turn concentrate the mind on data structures.

Within the Phase repository there is no defined sequence for the creation of entities. Where entities have a relationship, the relationships can be defined automatically. The level of detail required for entities is set to allow the minimum of information to be entered to create an entity with the fuller details being filled in at a later stage. The supporting Phase CASE tools therefore support opportunistic development.

8.4.4 Note Making

Note making is the most unstructured method of specification. It has the advantage of unconstrained syntax but the disadvantage of ambiguity, limiting its use as a specification medium.

The Phase repository system allows free format notes to be attached to any entity. These notes can be 'tagged on' at any time during the design process. The notes are not used as part of the formal 'specification' however they are available to add clarity to some aspects of the design. In some circumstances, these notes are available as part of (and can be edited via) the on-line help in the prototype of the product. This allows the designer to record notes easily during a prototype review with users. The notes are automatically linked to the relevant entities within the repository which are being simulated at the time.

8.4.5 Summary

When compared to the observations of Davies and Castell, it has been shown that the Phase paradigm is effective as a software designers productivity tool.

8.5 Phase as a Software Project Management System

The complete Phase project management system is described in Appendix B. In literature, there are considered to be three essential criteria for a project management system [Daily92]:

- The recognition of process milestones
- Auditability
- Team Development

8.5.1 The Recognition of Process Milestones

Often regarded as the most important requirement for a project management system is the ability to determine 'milestones' against which progress can be reported. These milestones should be in the form of 'deliverables'. Typically these are linked with 'progress payments' in a commercial situation.

The Phase system recognises four main 'deliverables' in the system. These are :

- The application overview (or Project Definition of Scope of Supply)
- Prototype Specifications
- Technical Specifications
- Finished Programs

These are described fully in appendix B.

8.5.2 Auditability

Any Quality System must be fully auditable [Daily92]. This is the basis for many of the general IT standards that are emerging eg TickIT, AQAP, ISO9001 etc. Auditability tends to be more associated with tracing the source of problems and the ability to replicate a standard, however 'good' that standard may be. Generally an auditable system contains accurate documentation in the form of a project log, with appropriate forms requiring 'signing'.

The Phase system has automatic logging which is described fully in Chapter 5. This logging provides a complete audit trail of every change made in the system. The QIR form, also described in Chapter 5, satisfies the requirements of BS5750 and the TickIT standards

8.5.3 Team Development

Large programming systems requires development teams. The essence of good team building is communication between team members. For software, due to its invisibility, communication of concepts is not easy. Traditionally, diagrams such as data flow diagrams, structure diagrams etc act as 'blue prints' for the software.

The Phase system has a number of diagrams as described earlier in this chapter. This makes communication of a Phase specification manageable.

8.5.4 Summary

When compared to the criteria given by Daily, it has been shown that the Phase paradigm is effective as a software project management system.

8.6 The Disadvantages of the Phase System

There are a number of disadvantages to developing with the Phase development strategy. These are summarised as :

- Changing entities in a prototype changes the actual program
- 'Clever' screen displays cannot be created
- It is not possible to rebuild previous versions of programs
- There is a maximum finite size of programs
- Programs require a large of amount of computing resources

8.6.1 Close Relationship between Prototypes and Programs

Earlier in this chapter it was observed that there is a very close relationship between a prototype and the target system. This was viewed as an advantage. This is also a major disadvantage.

Specifying changes to software requires changing the entities within the Phase repository which in turn updates the execution of the prototype. However, the 'current' version of the program may also use the same entities during its execution, the problem arises when an existing entity is altered for a change in specification, the functionality of the existing program is altered.

The simplest example of this is the addition of a new option on a menu. Creating the appropriate flow of control node within the repository will include this option on a prototype menu however, as menus are formed at run time by the target program, this option will also appear in the 'latest release' of the program.

A more significant example is where screen definitions are updated, again the screen layouts are referred to at run time of a program. The change to the specification may be to include new data items on a screen, this will cause the existing program to terminate abnormally.

In practice this requires careful timing of specification changes which cannot take place when a program may be involved in a maintenance release. This can be a significant problem. Incorporating version control would be a major contributor in eliminating these problems and is discussed below.

8.6.2 Inflexible Screen and Flow of Control Structure

All Phase applications have a similar structure which gives a consistent look and feel to the software. This is very advantageous from a user acceptance point of view as users who are familiar with one program can easily learn to operate new programs.

There are instances when the rigid structure of screen design and flow of control options can be limiting. For example, screens cannot be altered based upon the contents of previous data entered, the layout of a screen is fixed. This results in screens which may be over-complex with blank fields which are not relevant in some instances.

There are instances when an overlapping windows user interface is not appropriate for an application from a speed an simplicity point of view. One example of this was a Phase Till interface program which uses a computer as a point of sale till. The overlapping windows interface resulted in a program which was overcomplex for use on a shop counter with

relatively untrained staff. In practice, the Phase till system was replaced by a non-Phase till program.

8.6.3 Inability to Build Previous Software Versions

The Phase repository structure includes only the latest version of any entity. An entry is made in the 'entity change log' each time an entity is changed and, as discussed in chapter 5, the reason why this change was made is recorded. The previous attributes of the entity are however lost. This has the result that, should a new design be inappropriate it is impossible to retrace the design back to its original position. It is also not possible to have different software 'builds' of previous releases of a product. This can make replication of a users 'bug' very difficult to achieve without keeping run-time copies of all released programs.

Incorporating version control was only omitted due to fact that initially the Phase method was classed as experimental and therefore not a priority requirement. As Phase now has commercial implications the priority of this requirement is such that it will be implemented in the near future.

8.6.4 Maximum Finite Size of Programs

The structure of a Phase program requires a target language which can support separately compiled procedures linked into a single executable program file. This intrinsically limits the size of a program to constraints within the target language. Using the two different CASE tools, each with its own target language, these limitations have been reached for a number of programs. The effect of this can be reduced with a better split of functionality between programs, particularly where programs are used together as a 'suite'.

8.6.5 Computer Resource Usage

Phase programs require a subset of the Phase repository to be available during run-time. This imposes an overhead in terms of file-handles and execution efficiency for executable programs. It is observed that much of this dynamic run-time access could be circumvented as once a system has been configured the fact that additional configuration is available is superfluous

8.6.6 Summary

It has been shown that there are a number of recognised disadvantages to the Phase paradigm, however none of these can be considered critical. It is the intention to continue development with the Phase paradigm to overcome some of the disadvantages discussed with further maintenance.

8.7 Conclusion

When compared against literature, the Phase paradigm matches all the desirable features of all four different categories. It can be considered therefore as a serious contender as : a requirements analysis tool, a specification representation system, a software designers productivity tool and a software project management system.

Chapter 9

Summary

9.1 Introduction

Brooks suggested that :

> *"altering the Software Engineers attitude to change, from being an annoyance to accepting change as a way of life, would be a significant step in delivering quality systems".*

I suggest that the Software Engineers attitude towards 'change' is directly related to the methods and tools available for designing and creating software. I also suggest that there are two (perfectly reasonable) major factors for a 'bad attitude' :

- A sudden change in requirements can instantly make days, weeks or perhaps months of 'technically perfect' hard work suddenly become redundant
- Software that has been designed for a specific goal and written 'as a seamless work of art' is torn apart and stitched together to satisfy some change of requirements. This inherently leads to a detrimental effect on the quality of the software

'Bad attitude' leads to unsatisfaction. Unsatisfaction leads to staff turnover. Staff turnover leads to the disappearance of staff experience. Loosing this experience is costly for the commercial software developer.

9.2 The Nature of Requirements and Change

This thesis started by examining the nature of requirements. It states that requirements can be split into Requirement for the Software (the 'User Requirements') and Requirements of the Software (the 'Technological Requirements'). It has been studied and reported how and when these sets of requirements change.

The first stage for control, is to monitor. The monitoring of change and the effects of change were studied for a period of five years. As a result, a system for designing software which would be more resilient to the detrimental effects of change, was created. The principle behind this system was :

- Identify the state of the 'components' in a system
- Monitor and record the changes to the component states

This system is called Phase. The Phase system cannot exist without CASE tools which provide a means for implementing the theories.

9.3 The Phase Paradigm

The Phase paradigm, presented in chapter 5, uses a central repository to store information relating to the design of a system using five simple types of entities. These entities reflect : The flow of control , the user interface, the data storage, the functionality, the fifth being an entity which links the previous four together. This repository becomes the 'specification'. This specification can be executed as a 'prototype'. This easy-to-build prototype allows users to see exactly how the resultant software will look and feel.

Experience demonstrates that typically a user will require at least three attempts at refining requirements and each attempt is based upon actually using the results from the previous attempts. Without prototyping, this implies that at least two full systems will be 'thrown out' (by which time the Software Engineer is becoming frustrated and upset), using this prototype technique provides the same effect for the user without the same effect on the Software Engineer.

The same specification which is demonstrated by the prototype is used to prepare the actual program. The Phase paradigm is ideal for automatic code generation, translating the specification into executable software. This is implemented using the CASE tools.

9.4 Maintaining 'Experience'

The Phase paradigm attempts to automatically record the design decisions made by developers during the lifetime of the development activity. This information can be considered as capturing the 'experience' of these designers. By extracting and analysing this information, the 'experience' can be retained, long after a turnover of staff, to be passed to future developers. This is presented in Chapter 6.

9.5 A Tried and Tested Theory

The Phase paradigm and a number of its associated CASE tools are actively being used for commercial development of software. This provides a real 'test bad' for obtaining results. The level to which Phase software has been modified, especially after its initial release to users, is presented in Chapter 7. This demonstrates that there is a high degree of resilience to the detrimental effects of change for Phase applications. It does not remove completely, all the 'bad' effects of change. It is also possible to create 'bad' Phase programs as well as 'good' Phase programs.

9.6 An Appraisal of Phase

Chapter 8 presents an appraisal of the Phase paradigm. This is in relation to Phase as :

- A requirements analysis tool
- A specification representation system
- A software designers productivity tool
- A software project management system

This chapter also includes a list of points where the Phase paradigm and its CASE tools could be improved.

9.7 The Phase Repository and Project Management Techniques

A detailed statement of the Phase Repository is given as an Appendix. Also given is a set of working practices to provide an explanation of a 'Phase Program Lifecycle'. This provides a step-by-step guide to project managing a Phase development.

9.8 Conclusion

This chapter has summarised the aims and contents of this thesis. The next chapter presents the conclusions drawn from the period of research. Many of the conclusions have been previously presented earlier in this thesis.

Chapter 10

Conclusions

10.1 The Contribution of this Thesis

This thesis has proposed a paradigm for developing computer software called Phase. This paradigm is designed to tackle one of the highlighted essential elements of software development often ignored in traditional development strategies: that requirements will change. Consequently, programs designed using this paradigm are more resilient than traditional programs, to the detrimental effects of change.

The reasons why Phase programs are more resilient to change is because they are structured in such a manner that they combine many of the recognised properties of 'good programming practice'. These are :

- Data Driven techniques allow for flexibility and 'customisation' without requiring programming changes. This dramatically reduces the complexity of a program.
- Procedures are extremely highly cohesive and extremely loosely coupled. This limits the potential 'damage' of other procedures when changes are made.
- Prototypes of software are created very easily. The prototypes are very close to the look and feel of the resultant programs. This gives user the ability to 'throw away' as many copies of the prototype as required without wasting valuable software engineering resources.

- Diagrammatic documentation is automatically produced. This allows for visual representation of 'mental models' of a concept. These diagrams allow developers to share and communicate their mental models and ideas.
- All entities in the system are related in a natural 'hyperlink' form of structure. This provides the ability to easily check the implication of changes on other components in the system.
- Automatic code generation is performed translating the specification into target language code. This automatic code generation is based upon 'macro substitution' similar to the techniques used in many assembly code assemblers.

In addition to the above features which have been used for many years, a number of new techniques have been introduced. This is based upon the automatic recording of changes made to entities within the system and the way in which this information is used.

- All changes made to entities are recorded automatically by the system. In addition to who, and when, entities were changed, the reason why entities are changed is also recorded. This represents the design decisions taken by developers during the lifetime of the software.
- Using the logging of changes provides a method for easily checking work done by a development team. This is presented as the Quality Inspection Record document and inspection technique introduced in chapter 6.

It has been shown that this technique is successful for the commercial development of a class of software known as Interactive Business Information Systems. Many of the techniques could equally be applied to different classes of software.

10.2 Further Development

The Phase paradigm has been in existence in some form since 1986 as a direct result of a Stirling University Computing Science honours project. This led to the implementation of the Phase paradigm on a mini computer. During the past five years, technology forced a replacement of the mini computer with powerful PC networks. Technology on PC networks is

such that an inevitable change will take place where all applications will have to conform to the Windows GUI interface. A future enhancement will be to develop Phase applications adapted for this technology. How this will be attempted has been introduced in Chapter 7.

Chapter 8 highlighted some of the current disadvantage and problems with the Phase system and its supporting CASE tools. In particular, the problem caused by the prototype and the runtime elements of a program accessing the same version of the entities in the repository means that the ability to prototype additional functionality whilst maintaining a current version of a program is impossible. In order to increase the usefulness of this system, some form of version building will be required.

The analysis of the information logged as 'experience' is currently only presented to developers when it is explicitly requested and it requires a fair level of knowledge before this information can be classed as useful. Further enhancements to the system can include some form of 'expert system' which will analyse the information and automatically guide the developer through implications of changes.

10.3 The Attitude to Change

The question still remains :

What of the developer's attitude to change, is this still a problem ?

There have been twelve developers who have used the Phase paradigm for developing software. Of these twelve, six have stayed with the system since their first introduction. Of the remaining six who left, four have returned having not found a better system, the other two left for further education.

Some negative attitude to change still exists, a deeply bred culture takes years to change. Personally, however, I know that my attitude towards change is better. It's going to happen, be prepared for it : Users will be Users!

Bibliography

Agostoni88 — AGOSTONI, G. et al : "Managing software quality during the complete lifecycle". *1st European seminar on software quality* (Brussels, 1988)

Alexander92 — ALEXANDER, C : "An introduction for Object-Oriented Designers". *ACM SIGSOFT Software Engineering Notes* 1994 Vol 19 No 1 pp 39-46

Ambriola90 — AMBRIOLA, V; et al : "The evolution of configuration management and version control" *Software Engineering Journal*, Nov. 1990 Vol. 5 No. 6 pp 303-310

Balzer83 — BALZER, R. et al : "Software Technology in the 1990's : Using a New Paradigm," *Computer*, vol 16, no.11, pp 39-45, 1983

Basili75 — BASILI, V.R. et al : "Iterative Enhancement : A Practical Technique for Software Development, " *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 390-396, 1975

Benington56 — BENINGTON, H.D.: "Production of large computer programs" in *ONR Symposium on Advanced Program Methods for Digital Computers*, pp 15-27, June 1956

Blum91 — BLUM, B.I. : "Towards a uniform structured representation for application generation" *Int. J. Software Eng. Knowledge* Eng. vol 1, pp. 39-55, 1991

Blum93 — BLUM, B.I. : "Representing Open Requirements with a Fragment-Based Specification" *IEEE Transactions on Systems, Man, and Cybernetics*, vol 23, no. 3, pp.724-735, 1993

Boehm76 BOEHM, B.W. : "Software Engineering," *IEEE Transactions on Computers*, vol C-25, no. 12, pp.1226-1241, 1976

Boehm86 BOEHM, B.W. : "A Spiral Model of Software Development and Enhancements," *Proceedings of an International Workshop on the Software Process and Software Environments*, Coto Do Caza, California. March 1985, published as Software Engineering Notes, vol.11, no. 4, 1986, pp. 22-42.

Booch91 BOOCH, G.R.: 'Object Oriented Design with Applications'. Redwood City, California: Benjamin / Cummings

Brooks75 BROOKS, F.P. Jr.: 'The Mythical Man-Month' Reading. MA: Addison-Wesley, 1975

Brooks87 BROOKS, F.P. Jr.: 'No silver bullet: essence and accidents of software engineering', *Comput.*, 1987, (4), pp. 10-19

BTRL90 British Telecom Research Laboratories,UK. Rigby, P.J.; Norris, M.T. : "The Software Death Cycle". *UK IT 1990 Conference* pp 8-14

Budgen94 BUDGEN, D. : Software Design. Addison - Wesley

Christensen83 CHRISTENSEN, N; et al : "Prototyping of User Interfaces" in Approaches to Prototyping, Ed Budde,Kuhlenkamp, Mathiassen, Zullighoven

Chudge96 CHUDGE, J. ; FULTON, D. : "Trust and co-operation in system development: applying responsibility modelling to the problem of changing requirements." Software Engineering Journal. May 1996 pp 193 - 204

Cosgrove71 COSGROVE, J. : "Needed : a new planning framework," *Datamation*, 17, 23 (Dec. 1971) pp 37-39

Daily92 DAILY, K. "Quality Management for Software" NCC Blackwell Ltd, 1992 ISBN 1-85554-082-7

Davies92 DAVIES, S.P.; CASTELL, A.M. : "Contextualizing design: narratives and rationalization in empirical studies of software design', *Design Stud.*, 1992, 13, (4), pp. 379-392

Dawson95 DAWSON, C.W.; DAWSON, R.J.: "Towards more flexible management of software systems development using meta-models", *Software Engineering Journal*, Vol.10 No.3 May 1995

Fagan77 FAGAN, M.E. : "Design and code inspections to reduce errors in program development", *IBM Systems* Vol. 3. 1977 pp 182-206

Floyd83 FLOYD, C.: "A systematic look at Prototyping", in Approaches to Prototyping, Ed Budde,Kuhlenkamp, Mathiassen, Zullighoven

Gamma93 GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. : "Design Patterns : Abstraction and Reuse of Object-Oriented Design" in *Lecture Notes for Computer Science, ECOOP'93 - Object Oriented Programming.* July 1993

Giddings84 GIDDINGS, R.V.: "Accommodating uncertainty in software design," *Commun. Ass Comput. Mach.*, Vol. 27, no 5, pp428-434, May 1984

Harker93 HARKER, S.D., EASON, K.D., DOBSON, J.E. : "The change and evolution of requirements as a challenge to the practice of software." IEEE Int. Symp. on Requirements Change, 1993 (IEEE Computer Society Press).

Henderson86 HENDERSON, P. : "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Trans. on Soft.Eng.* Vol SE-12 No.2 Feb 1986

IEEE91 IEEE : IEEE Standard Glossary of Software Engineering Termology. In situ of Electrical and Electronic Engineers, inc, New York, USA, 1991. Revision and Registration of IEEE STD 729-1983

Junk88 JUNK, W.S. : More than just a diagramming tool. In *IEEE Software*, March 1988 Software Reviews pp 97.

Lee91 LEE, J : Extending the Potts/Bruns Model for Recording Design Rational, 1991.

Lehman80 LEHMAN, M.M.: "Programs, life cycles, and the laws of software evolution," *Proc. IEEE, vol. 68. no 9,* pp. 1060-1076, Sept 1980

Letovsky87 LETOVSKY, S. : "Cognitive Processes in Program Comprehension" *The Journal of Systems and Software* 7, 1987 pp 325-339

Liskov75 LISKOV, B.H.; ZILLES, S.N. : "Specification techniques for data abstraction," *IEEE Trans. Software Eng.*, vol. SE-1 pp 7-19, 1975

Littman87 LITTMAN, D., PINTO, J., LETOVSKY, S., SOLOWAY, E. : "Mental Models and Software Maintenance". The journal of Systems and Software 7, 1987, pp 341-355

McCracken78 McCRACKEN, D.D.: "The changing face of applications programming", *Datamation*, pp. 25-30, Nov. 15, 1978

NATO68 Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE Garmisch, Germany, 7th to 11th October 1968. Ed Peter Naur and Brian Randell, 1969

Parnas72 PARNAS, D.L.: "A technique for software module specification with examples" *Comm. ACM.* 15 pp.330-336, 1972

Parnas72b PARNAS, D.L.: "On the criteria to be used in decomposing systems into modules" *Comm. ACM.* 15(12), pp.1053-1058, 1972

Parnas79 PARNAS, D.L.: "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, Vol. SE-5, no. 2, pp. 128-137, Mar. 1979.

Parnas86 PARNAS, D.L.; CLEMENTS, P.C. : "A rational design process: how and why to fake it", *IEEE Trans.*, 1986, SE-12, pp. 251-257

Pfleeger94 PFLEEGER, S.L.; "Design and analysis in software engineering. Part 1: the language of case studies and formal experiments. *ACM SIGSOFT Software Engineering Notes* 1994 19(4) pp16-20

Potts88 POTTS, C.; BRUNS, G.: "Recording the Reasons for Design Decisions", *10th Internation Conference on Software Engineering*, April 11-15, 1988 pp 418 - 427

Potts89 POTTS, C.; BRUNS, G.: "A generic Model for Representing Design Methods" 1989, *ACM 0270-5257* pp 217-226

Pree94 PREE, W : "Design Patterns for Object Oriented Software Development. Addison - Wesley

Proteus93 PROTEUS : "Understanding changing requirements". Proposal to IEATP Safety-Critical Systems Programme, September 17. 1993

RE93 IEEE Int. Symp. on Requirements Change, 1993 (IEEE Computer Society Press).

Reeves95 REEVES, A.; MARASHI, M.; BUDGEN D.: " A Software design framework or how to support real designers", *Software Engineering Journal*, July 1995 pp 141-155

Ross77 ROSS, D.T.; SCHOMAN, K.E.Jr : "Structured Analysis for Requirements Definition", *IEEE Trans. Soft. Eng.* Vol. SE-3. No.1, Jan. 1977

Rowen90 ROWEN, R.B. : "Software project management under incomplete and ambiguous specifications", *IEEE Trans.*, 1990, **EM-37**, (1), pp. 10-21

Royce70 ROYCE, W.W.: "Managing the development of large software systems: Concepts and techniques," in *WESCON Tech. Papers*, Aug 25-28, 1970, pp. A.1 1-9

Shaw96 SHAW, M., GAINES, B. : "Requirements Acquisition". *Software Engineering Journal*, May 1996 pp 149 - 165

Simon69 SIMON, H.A.: "The Sciences of the Artificial." *Cambridge*, MA:MIT Press, 1969.

Smith72 SMITH, J : "Economics : A first course" *Oxford University Press*, 1972

Sommerville93 SOMMERVILLE, I.: "Software Engineering" Addison Wesley 1993.

Swartout82 SWARTOUT W.; BALZER R.: "On the inevitable intertwining of specification and implementation." *Common. Ass. Comput. Mach.*, Vol. 25, no 7. pp. 438-440, July 1982.

Takahashi95 TAKAHASHI, K., OKA, A., YAMAMOTO, S., ISODA, S. : "A comparative study of structured and text-oriented analysis and design methodologies" *Journal of Systems and Software*, 28: 49-58. 1995

Williams88 WILLIAMS, L.G.: "Software Process Modelling : A Behavioral Approach" , *10th International Conference on Software Engineering*, April 11-15, 1988, pp. 174-186

Appendix A

The Phase Repository Structure

This appendix contains the definitions for all the entities in the Phase repository. Each table is individually listed with its contents. A data table diagram is included at the end to show how each of the tables are related.

FMACTION		Each record contains a step in the life-cycle model
Field Name	**Type**	**Description**
ACTION	C2	Step number
ACTION_TYP	C1	Action or Deadline
ACTION_DESC	C30	Description

FMDBASE		Each record contains the definition for a data table (excluding data items)
Field Name	**Type**	**Description**
DBASEID	C10	Internal identification number
DBASENAME	C8	Name of data table
DBASEDESC	C60	Description
INXFILE01-15	15C8	Filenames of index files
INXKEY01-15	15C60	Index expressions
DATE	Date	Date last modified

FMDBITEM		Each record contains a link between a data table and a data item
Field Name	**Type**	**Description**
DBASEID	C10	Internal identification number of a data table
ITEMID	C10	Internal identification number of a data item
ORDER	N3	The order the item appears in the table
DESCTEXT	Memo	Description

FMGLOBAL		A single record table containing the configuration parameters
Field Name	**Type**	**Description**
PROGRAM	C40	A descriptive name for the module
COMPANY	C40	Company name of development company (used for report headings)
MACROPATH	C30	Pathname for algorithm definitions
FDSLIBPATH	C30	Pathname for library algorithm definitions
GENPATH	C30	Pathname for generated source code
USERNAME	C8	Username of application supervisor
NODEID	C10	Next node identification number to be allocated
PROCID	C10	Next procedure identification number to be allocated
MACROID	C10	Next algorithm identification number to be allocated
ITEMID	C10	Next data item identification number to be allocated
DBASEID	C10	Next data table identification number to be allocated
REQREF	N6	Next RPU reference to be allocated
RELEASE	C8	Current release number
DATE	Date	Date record last modified
COL_HEAD	C44	Colour for the user interface 'header'
COL_FOOT	C44	Colour for the user interface 'footer'
UTILITY	C8	Last data table upgrade utility executed
APPLIC	C2	Two character mnemonic for the application for validation

FMITEM		Each record contains a data item definition
Field Name	**Type**	**Description**
ITEMID	C10	Internal identification number
ITEMNAME	C16	Name of data item
ITEMDESC	C70	Description
ITEMTYPE	C1	Type of Item (Character/Numeric/Logical/Date/Memo)
ITEMLENGTH	N4	Length of Item
ITEMDECPL	N2	Decimal Places (Numeric Items)
PICTURE	C60	Standard Data Input Template
VALID	C60	Standard Data Input Validation Procedure
DATE	Date	Date last modified
DESCTEXT	Memo	Description

FMLINK		Each record contains a link from a node to a child node
Field Name	**Type**	**Description**
NODEID	C10	Internal identification number of the parent node
CHILDID	C10	Internal identification number of a child node
OPTIONNO	N2	Option number
FMLOG		**Each record contains a single entity modification**
Field Name	**Type**	**Description**
DATE	Date	Date the change was made
USERNAME	C8	Username of the developer making the change
LOG_SECT	C8	Type of entity
LOG_NAME	C16	Name of entity
REQ_REF	N6	Pointer to the 'why' table FMREQEST
LOG_REMARK	C24	Description of type of change eg Created/Modified/Deleted etc
TIME	C8	Time the change was made
ID	C10	Internal identification number of the element to which the record is associated

FMMACRO		Each record contains a pointer to an algorithm definition
Field Name	**Type**	**Description**
MACROID	C10	Internal identification number
MACRONAME	C16	Name of the algorithm
LIBRARY	C1	Library routine or Application only
DESC1-2	2C56	Description
PARAM01-10	10C10	Parameters passed to algorithm at code generation time
DATE	Date	Date record last modified
LOCK-FLAG	C1	Lock flag to prevent multiuser editing
LOCK-NAME	C8	Username of person locking algorithm

FMMODULE		Each record contains an entry for a SubModule (Used for the manual print)
Field Name	**Type**	**Description**
MODORDER	C2	Logical order for implementation
SUBMODULE	C7	Name of submodule
MODTITLE	C60	Chapter Title
MODTYPE	C10	Type of submodule (Data Entry/Report/Enquiry)
DESCTEXT	Memo	Description
CHAPTER	N3	Chapter number (when printing the manual)

FMNODE		Each record contains a flow of control node
Field Name	**Type**	**Description**
NODEID	C10	Internal identification number
NODENAME	C16	Name of flow of control node
DESCTEXT	Memo	Description
PROCID	C10	Internal indentification number of the called procedure
SELECTTYPE	C8	Selection type for options (Menu/Softkey/Fserial/None)
LABEL	C26	Menu or command select label
TITLE	C25	Identification name
COLOR	C20	Colour of menu
EXITID	C10	Internal identification number of the flow of control node used when exiting
HELPLINE	C60	Short message required for onscreen help
DATE	Date	Date record last modified
ACCLEVEL	C26	Security access level

FMONLINE		Each record contains an online tutorial
Field Name	**Type**	**Description**
HELPCODE	C4	Tutorial indentification number
FLAG	C2	A grouping field
QUESTION	C140	Title of the tutorial
ANSWER	Memo	Contents of the tutorial

FMPROC		Each record contains a procedure entity
Field Name	**Type**	**Description**
PROCID	C10	Internal identification number
PROCNAME	C16	Name of the procedure
OVERLAY	C8	Name of source file when generating source code
LEVEL	C1	Flag indicating if this is a configurable procedure or internal procedure
DIRTYFLAG	C1	Set true if any entity used by the procedure has been edited, cleared when generated
FASL	C200	Prototype definition command line
MACROID	C10	Internal identification number of the algorithm definition
MACRO	C200	Name of the macro and actual parameters as a command string
DESCTEXT	Memo	Description
DATE	Date	Date record last modified
LOCK-FLAG	C1	Set true to lock entry if it is being edited off-line
LOCK-NAME	C8	Username of developer locking the entry
ICD01-05	5C8	Username of developers responsible for developing the procedure

FMPROJ		Each record contains an entry in the planned development timetable
Field Name	**Type**	**Description**
ACTION-COD	C2	Code for the action
DLINE-CODE	C2	Code for the deadline
USERNAME	C8	Username of the person responsible for the action/deadline
DUE-DATE	Date	Date the deadline is due
COMP-DATE	Date	Date the deadline actually reached
DESCTEXT	Memo	Description or notes

FMREQEST		Each record contains the definition of a 'state'
Field Name	**Type**	**Description**
REQ_REF	N6	Request identification number (RPU)
REQ_DESC1-3	3C76	Description
DATE	Date	Date request raised
REQ_SOURCE	C10	Source of request (developer/client)
REQ_STATUS	C1	Current Status
REQ_PRI	C1	Priority
RELEASE	C8	Release Number for completed requests
REQACTION	C20	Work authorisation code
DESCTEXT	Memo	Release notes

FMSCRF00		Each record contains a screen item definition
Field Name	**Type**	**Description**
SCREENNAME	C10	Name of the screen
FIELDNAME	C16	Name of the field
ITEMID	C10	Internal identification number of the field
SOURCE	C8	Data table containing field
ROW	N3	Screen coordinate - row
COL	N3	Screen coordinate - column
LENGTH	N3	Screen length of field
PICTURE	C40	Actual data input template
VALID	C40	Actual data input validation procedure
SAYGETTYPE	C10	Screen version identifier - Display only or Input/Output
ITEMTYPE	C1	Type of data item
ITEMDECPL	N2	Data item decimal places
BLOCK	C2	Cursor order block
FLABEL	C30	Field label (used for automatic documentation)
FDATA	C30	Field data used on prototype of screens
DESCTEXT	Memo	Description
PALETTE	N1	Colour palette of field

FMSCRI00		Each record contains the definition for a screen
Field Name	**Type**	**Description**
SCREENNAME	C10	Name of screen
TOP	N2	Screen coordinate for window top margin
LEFT	N3	Screen coordinate for window left margin
BOTTOM	N2	Screen coordinate for window bottom margin
RIGHT	N3	Screen coordinate for window right margin
IMAGE	Memo	Screen image (graphics characters, field labels etc)
DATE	Date	Date screen last modified
MODE	N3	Screen mode (80x25, 80x50, 132x50)
DIRTYFLAG	C1	Set true if screen has been modified, cleared when generated

FMUSER		Each record contains a user for each developer with access to the module
Field Name	**Type**	**Description**
USER	C8	Username
ACCLEVEL	C1	Access Level (Programmer/Support Only/Project Manager)
REQ_REF	N6	Current RPU being edited

FMXREF		Each record contains a hyperlink cross reference between entities
Field Name	**Type**	**Description**
ID	C10	Internal identification number of the one side of the link
CHILDID	C10	Internal identification number of the other side of the link

Appendix A : Repository Data Table Structure Chart

The following chart show the relationships between the tables in the Phase repository as described earlier in this appendix. Note that the links from FMLOG and FMXREF have been omitted for clarity they link to the majority of the other tables.

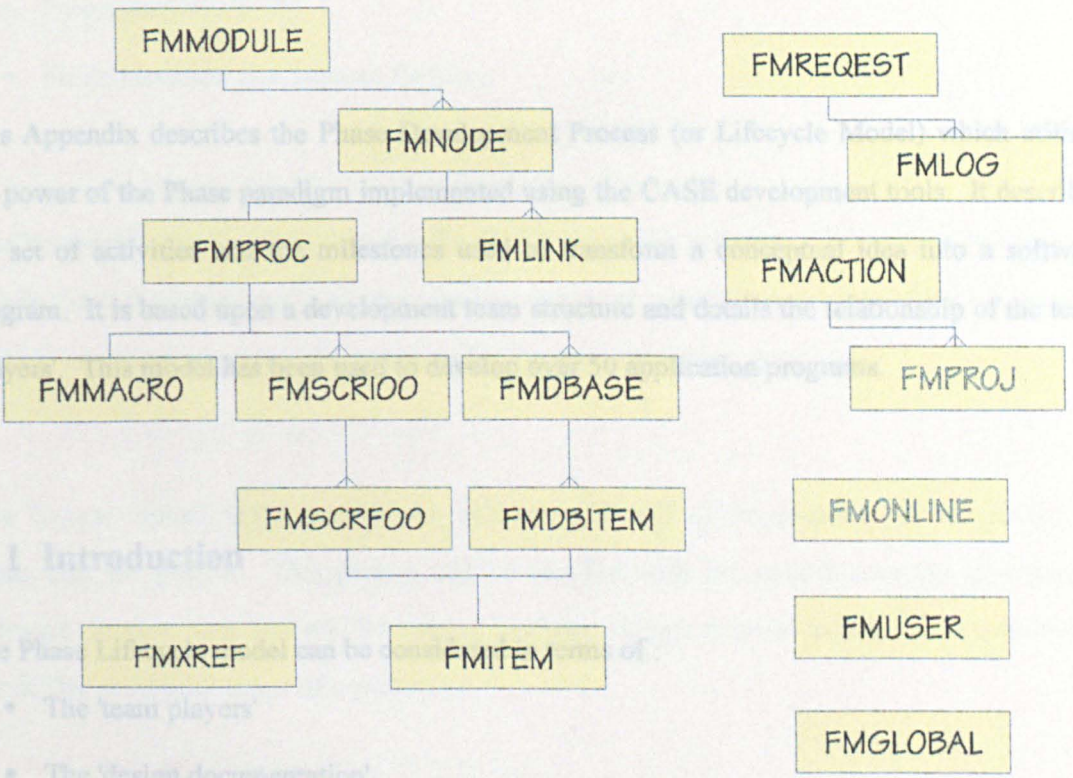

figure A : Database table structure chart

Appendix B

The Phase Development Process

This Appendix describes the Phase Development Process (or Lifecycle Model) which utilises the power of the Phase paradigm implemented using the CASE development tools. It describes the set of activities and the milestones used to transform a conceptual idea into a software program. It is based upon a development team structure and details the relationship of the team 'players'. This model has been used to develop over 50 application programs.

B.1 Introduction

The Phase Lifecycle model can be considered in terms of :

- The 'team players'
- The 'design documentation'
- The 'control documents'
- The 'Actions and Deadlines'

B.2 The Team Players

A 'team player' is a generic term for any person involved with the development of a Phase application. The possible team players are :

- Customer
- Liaison Contact
- Project Manager
- Project Designer
- Programmers
- Implementation and Support Engineer

Customer

The 'customer' is the generic term for the 'End User' of the software. This person, or group of people, have the basic 'need' and ideas for projects to increase their working efficiency. In most cases it can be assumed that the customer has limited computer appreciation.

Liaison Contract

The 'liaison contact' is the person who will act on behalf of the customer in formulating the ideas into the project. This person will be familiar with the overall concepts of computer software development and will be able to perform enough systems analysis to determine the feasibility and initial scope of a project.

Project Manager

The 'project manager' is the person who will assume overall responsibility of the project. This includes the definitive scope of the application, the maintenance of the development schedules and the quality assurance of the finished programs. The project manager will create the Application Overview (the list and documentation of the SubModules) within the CASE tool. It is essential that a project manager is totally familiar with the Phase development paradigm.

Project Designer

The 'project designer' will take the predefined scope of a program and create a prototype model of the software which will best implement the requirements. This prototype will be reviewed and refined until the design is accepted by the customer. The project designer is responsible for system testing of the application.

Programmer

The programmer will implement the details of the prototype program specification using the automatic code generators where possible. The programmer is responsible for the quality of the finished programs.

Implementation and Support Engineer

The 'implementation and support engineer' will configure and support the software for the customer. The implementation and support engineer will be responsible for all user documentation and training.

B.3 The Design Documentation

The 'design documentation' refers to the tangible components produced throughout the development process, representing the state of the design at various times. There are five major 'design documents' :

- Rough Notes
- Project Definition (Application Overview)
- Prototype Specification
- Technical Specification
- Finished Program

Rough Notes

'Rough Notes' are free format notes taken by the Liaison Contact during the numerous discussions with the Customer. They will include block database diagrams, rough screen and report layouts and general descriptions with data flow diagrams. Any number of tools can be used for these notes e.g. Work Processors, Screen Designers etc. There is no fixed format.

Project Definition (Application Overview)

This is a list of major functional elements in the system and created as a list of SubModules within the Phase CASE tools. This provides a 'scope' of work and is prepared as simple paragraphs of text.

Prototype Specification

This is the main tool used for communication between players in the project team. It includes every screen layout, database definition, flow of control information and report definition. A hardcopy version of the prototype is available.

Technical Specification

The 'technical specification' is a document automatically produced by the Phase CASE tools. It contains a complete definition of every entity defined within the repository.

Finished Program

The finished program is the program supplied to the customer. This includes all the appropriate documentation.

B.4 The Control Documents

The control documents are 'progress reporting' documents which provide a schedule and timetable for the project. The control documents are :

- Project Control Log
- Implementation Control Document (ICD)

- Status Report
- Request for Program Update (RPU Log)
- Quality Inspection Record (QIR)

Project Control Log

The 'project control log' includes general information about the project, the team players, a log of meetings, actions and deadlines (discussed below). It will always contain at least one current action with a deadline.

Implementation Control Document (ICD)

The 'implementation control document' is automatically prepared from the list of 'procedure entities' within the repository. It lists each procedure and its implementation status e.g. programmed, tested etc. It is used to provide a definitive 'status' of a program.

Request for Program Update (RPU Log)

This is used to monitor the support and ongoing development of an application after its initial release. It corresponds both to a 'wish list' for user requests and also a 'release log' of completed changes. The RPU log is explained in detail in Chapter 5.

Quality Inspection Record (QIR)

The 'quality inspection record' is produced by the programmer when completing a set of changes. This is used as an audit trail for logging changes made to entities within the repository. This is explained in detail, and an example form given, in Chapter 5.

B.5 The Actions & Deadlines

The following table represents the different phases that a development project will pass through. A phase is divided into stages and each stage has an action with a resultant deadline. This provides a method of monitoring an managing a project by providing tangible milestones. The actions and deadlines are explained in the remainder of this appendix.

Phase	Stage	Action	Stage	Deadline
A	1	Project Discussions	1	Project Initiation Meeting
	2	Initial Scoping	2	Project Investigation Meeting
	3	Appointment of Team	3	Acceptance of Project Definition
B	1	Prototype Specifications	1	Internal Design Meeting
			2	External Design Meeting
			3	Acceptance of ICD
C	1	Program Coding	1	Implementation Meeting
			2	Quality Assurance Meeting
			3	Program Beta-Test Release
			4	Program Release "C"
D	1	Program in Use	1	Project RPU Meeting
	2	Maintenance Programming	2	Maintenance Release

figure B.1 : Phase Lifecycle Table

At any point in time, the development of an application can always be defined in terms of an action stage associated with a deadline stage within the same phase. For example : Program Coding can be a current action, and it could have as a deadline any of the deadlines in this phase i.e. Implementation Meeting, Quality Assurance Meeting, Program Beta-Test or Program Release "C".

Note that the stages do not occur in a sequential manner and it is possible to move from stage D to stage B etc. It is also possible that different parts of a development may be in different stages and phases at the same time. For example, some aspect may be in program coding (Phase C) whilst another component of an application may just being developed (Phase B). The system therefore allows and encourages concurrent developments by different team members.

Stage
Project Discussions - Action Customer, Liaison Contact The Customer approaches the Liaison Contact with ideas. These ideas are discussed and rough notes taken. A feasibility study is undertaken and potential projects are conceived
Project Initiation Meeting - Deadline Liaison Contact, Project Manager The Liaison Contact approaches the appointed Project Manager and the rough notes are discussed. An approximate budget price is agreed
Initial Scoping of Project - Action Project Manager The Project Manager takes the rough notes and create the application overview in the Phase repository. The rough notes are translated into block diagrams and an application overview document.
Project Investigation Meeting - Deadline Liaison Contact, Project Manager, Customer The Liaison Contact initiates a meeting to discuss the overview. The customer may or may not be present depending upon the initial analysis performed and the complexity of the application.
Appointment of Project Design Team - Action Project Manager, Project Designer As the Application Overview is near completion, the Project Manager appoints a Project Designer and presents the application overview.
Acceptance of Project Definition - Deadline Liaison Contact, Customer, Project Manager, Project Designer The Liaison Contact arranges a meeting with the Customer and the Project Manager. The application overview is presented, discussed and finally accepted. This is the last involvement of the Liaison Contact with respect to the analysis details of the project. The Project Designer is introduced to the Customer.

Stage
Prototype Specification - Action Project Designer The Project Designer takes the application overview and rough notes and prototype specification. This will be produced in two parts. Part 1 showing screen layouts only, Part 2 showing field definitions
Internal Design Meeting - Deadline Project Designer, Project Manager The Project Designer arranges an internal meeting with the Project Manager to discuss the prototype specification
External Design Meeting - Deadline Project Manager, Project Designer, Customer The Project Manager arranges a project meeting with the customer and proposes the implementation. The prototype is reviewed with the hardcopy prototype document being updated by the Project Designer.
Acceptance of Prototype Specification - Deadline Project Manager, Project Designer The prototype specification is agreed by the customer
Acceptance of ICD - Deadline Project Manager, Project Designer, Programmer The Implementation Control Document is produced by the Project Designer and approved by the Project Manager. The programmer is introduced to the project
Program Development - Action Project Designer, Programmer The Programmer translates the Phase specification into programs. As each procedure is programmed the ICD is updated The Designer tests each procedure for conformance to the specification. As each procedure us tested the ICD is updated.
Implementation Meeting - Deadline Programmer, Project Designer The current development version of the software is copied into a test environment for the Project Designer to perform a system test. The QIR is printed upon acceptance.

Stage
Quality Assurance Meeting - Deadline Project Manager, Programmer, Project Designer The application is presented to the Project Manager who performs a second system test. Individual procedures can be checked against the Program Standard. The ICD is updated.
Program Beta-Release to Customer - Deadline Project Manager, Project Designer, Customer, Support Engineer The program is installed by the Project Manager or Project Designer and the Customer is trained in its use.
Program Use - Action Customer The program is used by the customer and a list of program alterations produced. These are logged in the RPU log by the Project Manager in the Phase repository.
Project RPU Discussion - Deadline Project Manager, Project Designer The RPU log is produced by the Project Designer and examined by the Project Manager. By consultation with the Customer a definitive list of modifications is created.
Program Acceptance Release "C" - Deadline Project Manager, Customer, Liaison Contact A program version is reached where the number of alterations allowed in a system is limited by the contractual agreement.
Maintenance Programming - Action Programmer This is called maintenance programming to indicate that it is done after the initial contractual agreement is reached, however in the Phase system, unlike conventional system, changes at this time are encouraged.
Maintenance Release - Deadline Customer The software is released to the Customer

Appendix C

Acronyms

This appendix contains a list of acronyms used in this thesis.

Acronym	Expansion
3GL	3rd Generation Programming Language
4GL	4th Generation Programming Language
BASIC	Beginners All-purpose Symbolic Instruction Code (Programming Language)
CASE	Computer Aided Software Engineering
EDS	Elite Development System
FDS	Foreman Development System
GUI	Graphical User Interface
IBIS	Interactive Business Information Systems
ICD	Implementation Control Document
NATO	North Atlantic Treaty Organisation
OO	Object Oriented
OS	Operating System
PC	Personal Computer
PH	Programmer Hour
PHASE	Philip Harwood's Approach to Software Engineering
QIR	Quality Inspection Record
RPU	Request for Program Update
TMS	Thom Micro Systems Ltd