University of Durham

School of Engineering and Computer Science
(Computer Science)

Cost Factors in Software Maintenance

John R. Foster

Ph.D.
1993

Abstract

Cost Factors in Software Maintenance

John R. Foster

This thesis addresses the problem of accounting for both the costs and the benefits of software maintenance in a commercial environment. It shows that maintenance can be regarded as an investment activity, and rejects the view of maintenance as simply a costly chore.

The objective is achieved by setting up a general model of maintenance, whose most detailed features are the technical actions which accomplish the task, but which also encompasses management controls and their financial consequences. Parts of the model are used to develop a formal description of the process of selecting change requests by priority and planning the timing of the software release.

The calculations of the model are implemented as a computer program, whose inputs are parameters describing a set of change requests and salient project details. The program is exercised with specimen data, and planning options and their outcomes are explored. Outcomes are expressed in terms of timescales and returns on investment. It is concluded that such analysis is not only desirable, but achievable in actual commercial projects.

Acknowledgements

I would like firstly to thank my supervisor, Malcolm Munro, for his guidance and encouragement throughout this project. Without his support, I could not have completed the project. I would also like to thank BT for their invaluable support and sponsorship of the work.

The management and staff of the research library at BT Laboratories have been tireless in their enthusiasm to help, and to them I record my deep thanks.

Many individuals have helped in ways that cannot be attributed directly by references in the text. Some have been my managers at different stages (and all actively supportive); some have provided their own bibliographies or insights through discussion; and a noble few provided help with the proof-reading. They are:

Colin Archibald, Andy Beasor, Keith Bennett, Roger Browne, Frank Calliss, Hilary Calow, Peter Cochrane, Bill Collins, Mel Colter, Simon Cooper, Nigel Coulter, Roy Everett, Nigel Fletton, Bob Frost, Ian Garrett, Bob Higham, Ann Horsey, Charles Jackson, Adrian Jolly, Darius Karkaria, Rachel Kenning, Hans Kiekuth, Ray Lamb, Bridget Leathley, Jan LeFèvre, Steve Lineham, Dave Lumby, Mark Norris, Fay Owston, Tim Pennick, Kearton Rees, Peter Rigby, Andrew Rombach, Sinclair Stockman, Alan Stoddart, Richard Storey, Mike Tilley, Nigel Titley, Bob Wachtel, Richard Warden, Ivan Warner, Paul Whittle, Phil Williams and Neil Winton.

Copyright

This work is dedicated to my wife Sheila, and my daughter Joanna.

Contents

Chapter 1

Introduction

1.1 Software and Software Maintenance

The design, development and subsequent maintenance of computer software are the realm of what is generally known as the discipline of software engineering. The accepted use of this term dates from 1968 when an historic conference was organised in Garmisch, Germany, under the auspices of the Study Group on Computer Science of the NATO Science Committee.

In a background note to the conference, it is recorded [Nau76, p. 5] that:

> *In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be used on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.*

Twenty-five years on, software retains its special reputation as one of the least disciplined of engineering subjects, despite its all-pervasive nature and the fact that the great majority of software developments are successful and productive. It is often pointed out, and fairly, that software engineering is a very young discipline when compared to others; but we would also stress two factors that have little to do with maturity:

Production Cost Ratios: The cost of an office building is almost entirely in its raw materials and construction, with design costs relatively small. Design errors can be enormously expensive to correct, and extra design effort to avoid them is intuitively cost effective. With software, the actual construction is carried out automatically by compilers, linkers etc. and the raw materials, insofar as they can be said to exist at all, consist mainly of storage space on magnetic or other media. Design represents almost the entire cost, and it is much harder to resist pressures to cut corners.

Mathematical Modelling: Whether the hardware product be an office building, a bridge or an aircraft, mathematical and other modelling techniques will be used to validate the design before construction takes place. The inevitably approximate nature of the modelling is acceptable because the final product is essentially continuous in its behaviour: small inaccuracies in design or construction will lead to small deviations in performance, and whole ranges of behaviour can be described by mathematics that is linear, analogue and well understood. With software, the smallest deviation of the product (a difference of a single bit) can easily lead to behaviour that

is drastically different from that expected. The mathematics that applies here is chaos theory, and the ability to model the behaviour of the product is severely curtailed.

After construction, both hardware and software products enter phases that are termed "maintenance," but here again there are strong differences. The great majority of hardware maintenance consists of the replacement of components that have failed through stress or age; actual design changes are likely to be few and limited in their scope. Hardware maintenance by and large attempts to restore original functionality, with a high probability in any particular case of success. Software maintenance, on the other hand, consists entirely of design changes; and the extreme sensitivity of the product's behaviour to tiny differences in implementation makes such changes intrinsically more likely to fail in unexpected and even catastrophic ways.

We do not, then, expect software engineering to evolve with time into "just" another engineering discipline, with the same attributes and predictability as others. It will continue to improve, but the scope for convergence is restricted.

1.2 The Thesis Position

1.2.1 Motivation for Thesis

The thesis came into being as the result of a wish on the one hand, to provide an element of formal description to the software maintenance process and on the other, to show that the formal description could lead to benefit in a com-

mercial setting. The author's past experience as head of a team engaged in the software maintenance of telephone switching systems played no small part in this motivation.

1.2.2 Statement of Problem

All organisations make investments in the tools and equipment needed for them to function, and for most companies, computer programs make up part of that inventory.

Investments in hardware (machine tools, buildings, vehicles etc.) are costed and accounted for under rules that are well understood. Capital investments are shown in company accounts as asset values to which depreciation applies, and investment decisions are based upon comparisons with the estimated earning power of the equipment involved. Maintenance costs are generally taken into account in these initial calculations, and regarded thereafter as relatively fixed expenses that have already been allowed for. Future design changes are not taken into account unless they can be clearly predicted; rather, they will be treated as though they were fresh investments when the need arises.

Investments in software are, at the development or purchase stage, amenable to similar calculations (although the costs are rarely reflected in company asset sheets). The provision for maintenance, however, meets with serious difficulties. Routine maintenance of the predictable kind is entirely absent, yet it is known from past experience that for the average product, software maintenance will cost as much again as development. This maintenance is the accumulation of

many unpredicted design changes. Often the cost may be ignored altogether in the initial planning; if it is not, the alternative is often seen as being to assume some overall average figure, and to enter that into the cost calculations. Either way, when the design changes are later proposed, there is no foundation for their cost-effectiveness to be assessed in the same manner as for hardware.

1.2.3 Statement of Contribution

The thesis proposes and demonstrates a method for the assessment of decisions to undertake software maintenance tasks in response to change requests. The assessment method uses financial criteria, which means that alternative investments may be considered and compared. The method derives from a formal description of the maintenance process, and is shown to be capable of implementation for potential commercial use.

1.2.4 Criteria for Success

The thesis takes as its basic premise that decisions taken during software maintenance should be informed by financial criteria, so that investment in maintenance can be justified not only in itself but against other potential applications of funds.

In order to meet its aims, the thesis must:

1. Describe the maintenance process with the aid of a quantified model that provides financial analysis of the consequences of proposed maintenance actions

2. Show that the use of the model aids decision making both within a maintenance project and in the comparison of investment values between projects

3. Show that the model can be implemented on a computer and that the implementation has the potential of practical commercial use.

1.3 Thesis Overview

Chapter 2 begins with some general remarks on the software maintenance process and what is known (or believed) about it. A definition of software maintenance is established, and common attitudes towards it are discussed. The current state of knowledge is briefly reviewed, followed by discussion of current research efforts.

Chapter 3 concentrates on models that have been proposed for the maintenance process, showing the variety of approaches that have been adopted.

Chapter 4 gives a qualitative description of the model that has been developed for the work of the thesis. It is divided into seven levels; one (the team level) is central to the thesis, but the existence of the others is important in demonstrating the model's ability to span from technical detail to company asset base.

Chapter 5 expands the model's description, but now concentrating on a quantitative description of the team layer and the parameters associated with it. Every parameter and derived quantity is described in this chapter, which thus comprises the complete formal model on which the thesis is based.

Chapter 6 returns to a more qualitative style and describes how the calculations of the previous chapter may be manipulated in the evaluation of maintenance options.

The calculations are well suited to a computer program and Chapter 7 describes the implementation that has been developed, together with its limitations.

Chapter 8 then brings together the formal model and its implementation, by setting up a number of maintenance scenarios and showing how they are evaluated. The scenarios are based on an exercise carried out on an actual maintenance project, but the details have been somewhat disguised in the interests of commercial confidence.

Finally, Chapter 9 concludes the thesis by summarising the results from the previous chapters in the light of the criteria for success, describing the opportunities for further research, and listing the conclusions of the thesis as a whole.

Chapter 2

Software Maintenance

2.1 Introduction

Different people hold different views on what is meant by the term "software maintenance", and a few even refuse to use the term at all. In this chapter we will begin by examining the main competing definitions of the term, and establishing the definition that will be used during the rest of the thesis. Section 2.3 considers people's attitudes towards maintenance, both historically (including discussion of the reluctance of many to use the term at all) and in the light of more recent and positive changes. Section 2.4 offers some perspectives on maintenance, citing the growth of research and the great range of project sizes. Section 2.5 summarises present knowledge about the overall costs of software maintenance. Section 2.6 then looks at some of the research efforts that have led to proposed and actual improvements in maintenance operations. In Section 2.7 we consider the motivation for research and the importance of finance

in it, establishing the motivation behind the work in this thesis.

2.2 Definitions

There are at least three definitions of software maintenance that are in widespread use. Two of them are frequently used in industry, while the third is preferred by researchers and is also the official definition according to the IEEE (The Institute of Electrical and Electronics Engineers, Inc.)

The first and perhaps most restrictive view is that maintenance is the activity of correcting errors in operational software [Fro85]. It is used in some areas of data processing, where separate teams of people work on the same programs: one team will be adding functionality to a base release, while another team will be responding to user problem reports and building new versions of the programs on a regular basis. The advantage is that each team is geared to its particular timescales; a disadvantage is that changes must be implemented twice, if the later enhanced release is to reflect user requests already granted.

The second view is also found in data processing, and is related to the first. Instead of distinguishing maintenance activities by the reason for change to the programs, it classifies as maintenance any change that is expected to take less than a certain amount of effort. The actual threshold varies between organisations, but typical values range from a few days to (less commonly) a few weeks or even a year [FH79]. In a variant of this definition, it may also be decreed that all corrective actions are classified automatically as maintenance — a combination of the two views.

The two views may be described respectively as the technical and the economic approaches. What they have in common is that each provides a way of avoiding the calculation of detailed costs and benefits for individual changes: instead, overall levels of activity are restrained by fixed budgets, and the goal is to achieve as much as possible within those budgets.

Under either view, it is apparent that user requests which are classified as maintenance are acted upon more quickly than those that are classified as development. Users are not slow to react to this and may well try to turn the distinction to their advantage, for example by disguising enhancement requests as requests for the correction of errors [Swa79, Gre84]. A further corollary is in planning the activities: maintenance requests are typically given much less scrutiny than enhancement requests before they are authorised. This makes them even more attractive to users as a means of getting changes made, but it also reduces the emphasis on seeking cost-effectiveness in maintenance.

These reasons are enough to make the commercial definitions unattractive from the point of view of research study, and this is given added force by the fact that the process of making any change to operational software is for the most part independent of the reason for that change. Tools and techniques that benefit error correction also benefit the making of enhancements, and the risks inherent in each activity are the same. We will therefore adopt a comprehensive definition: that software maintenance is the set of activities involved in making any change to operational software, for whatever reason. This is consistent with the usual interpretation of the standard IEEE definition [IEE93]:

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

2.3 Attitudes

Although people's attitudes towards software maintenance are changing, it has long been regarded as a second-class subject and has only comparatively recently been seen as a respectable subject for the attention of researchers. The widespread "quick-fix" view must take much of the blame for this; coupled with the lack of attention to cost-effectiveness, it leads quickly to a view among management that software maintenance is an expensive chore rather than a productive and profitable activity.

This misperception has had several side effects. In their pioneering survey of software maintenance [LS80], Lientz and Swanson reported that senior managers may add to the problems of maintenance if they resist involvement in it through their perception of the effect on their own recognition and career advancement. That such attitudes exist is supported by other surveys as well as by direct observation; for example, a survey in 1986 explored whether managers' attitudes towards maintenance were favourable, neutral or unfavourable. Only 17% of the 158 respondents viewed it favourably, with 38% being neutral and the other 45% taking an unfavourable attitude [Cha86]. One can only speculate on the effects of this on their maintenance staff.

One result of negative attitudes has been the renaming of activities that we

would classify as software maintenance. A selection culled partly from the literature but mostly through conversations between the author and maintainers and their managers is as follows: further development; production programming [Can72]; current engineering; Phase 2; post deployment software support (this one from the US military); software continuation engineering (advocated in [Jon81] but not seen in practice by this author); post-release development; post-design services; phased development; installed system development; software sustaining engineering. None of these terms has seen widespread use, perhaps because of a feeling that if any did it would attract the same reputation and lead to a further search for synonyms.

If the above summarises the traditional attitudes towards the subject, one can at least point to a growing positive interest as the business and economic importance of software maintenance gains recognition. From a negligible base, we shall see in the next section the sharp rise in the volume of published results over the last decade or so.

2.4 Perspectives

2.4.1 Publications

The origins of the growing recognition of software maintenance may be traced back to the decade between 1970 and 1980, in which numerous papers drew attention to the fact that maintenance is a greater consumer of resource than development. Estimates of the actual proportion of software expenditure devoted

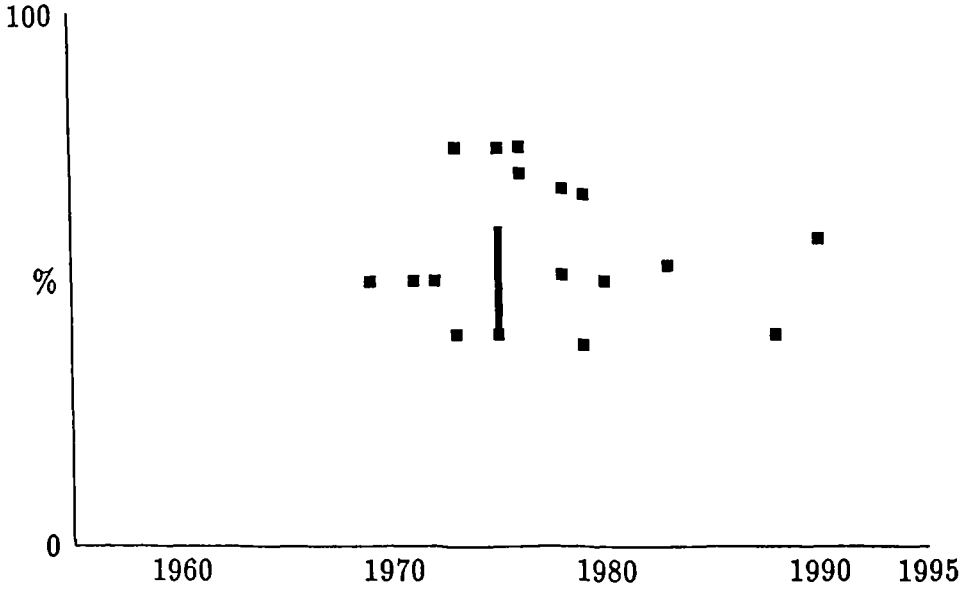

Figure 2.1: Published estimates of maintenance costs

to maintenance varied between about 40% and 75%, with general agreement that whatever the actual percentage, it was rising steadily and could be expected to continue to do so. Figure 2.1 illustrates the published estimates: each point shows the date of publication, and the estimated percentage of software expenditure devoted to maintenance. The references from which these figures were taken are given in Table 2.1. As may be seen from the figure, the 1970–80 decade saw the sudden onset of such estimates, falling to a low level after 1980 (by which time, we suggest, the message had been put across).

After about 1975, the first responses to this concern began to appear in published papers. As a measure of the effect, a search of the INSPEC database of publications was carried out. The two key phrases "software maintenance" and "software development" were used, and the number of papers in each year in

Year	Source	Maintenance Costs
1969	[Rig69]	Possibly as high as 40% to 60% for most companies who have had a computer systems effort for a number of years
1971	[DSA71]	About 50% of programming expenses
1972	[Can72]	About 50% of programming expenses for most business users
1973	[J. 73]	About 40% of programming resources
1973	[Dat73]	75%
1975	[Mil75]	75% of DP personnel are occupied with maintenance
1975	[Kha75]	Up to 40% of programming resources
1975	[Bro75]	40% or more of cost of development
1975	[BO75]	40–60% of software costs
1976	[Els76]	75% of costs
1976	[Boe76]	Probably about 70% of overall cost
1976	[Liu76]	In many organisations, at least 70% of time of analysts and programmers
1978	[LST78]	51% of time of systems and programming personnel
1978	[Zel78]	67%
1979	[MW79]	66%
1979	[FH79]	38%
1980	[LS80, p. 153]	Often held at around 50% as a result of deliberately freezing enhancement work
1983	[Gui83]	53% of software costs
1985	[Boe76]	(Prediction) 60% of all (hardware and software) costs
1988	[Mor88]	40% of software costs
1990	[NP90]	58% of software costs

Table 2.1: Published estimates of maintenance costs

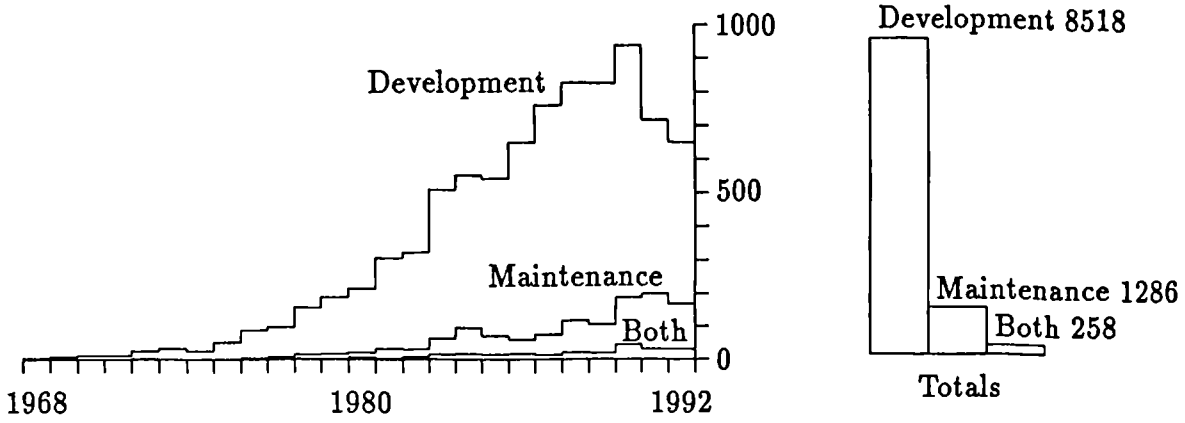

Figure 2.2: Published papers referring to software development, software maintenance or both terms. On the left are the numbers of papers by year; on the right are the accumulated totals.

which either or both phrases appeared in the title, the index keys or the abstract were recorded. Figure 2.2 shows the resulting graph on the left, with the totals as a histogram on the right. Each part of the figure shows the discrepancy between the respective levels of publishing activity, but it can at least be said that after about 1980 the numbers of maintenance papers have been such as to establish a firm presence in the literature.

If the decade to 1980 can be described as the decade of awareness, the decade to 1990 can at least be described as the decade of initial response.

2.4.2 Size of The Task: Foster's Metric

Not all maintenance projects are created equal, and differences in scale and in application can demand substantial differences in approach, from the one-person "team" to the dedicated department and from the casual to the highly formal.

To illustrate differences in project size, it is customary to refer to the number of lines of code that the project contains. The line count is an approximate measure in that many different definitions are possible, but here we are interested in gross differences only and will tolerate that approximation and more. It has another disadvantage, however, in that it does not encourage visualisation of the quantity described. The difference between a 100,000 line project and a 1,000,000 line project is considerable, but to most people the numbers convey little real impression of the magnitude.

In response to this latter problem, this author introduced at a conference presentation in 1987 [FM87] an alternative expression of the line count metric, which has become known as Foster's metric. Its formula is simple if anachronistic: it is the length of line printer paper required in order to print the entire program. The basic unit is the mile in countries that use that measure of distance, or the kilometre otherwise. Line spacing at the line printer standard of 6 lines per inch is assumed, which with a little allowance for page margins gives conversion factors of 400,000 lines per mile, or 250,000 per kilometre.

On this basis, one tenth of a mile represents a baseline where programs are of more or less human dimensions. In 1980, published survey results indicated that the average application program in data processing was about 23,000 lines of code, or 0.06 mile, and the same survey indicated that the average member of maintenance staff was responsible for about 38,000 lines (0.1 mile). From informal observations by the author, about twice this baseline would be the extreme limit of what one person could handle alone, and such a person would need to start with a high degree of knowledge of the system being maintained.

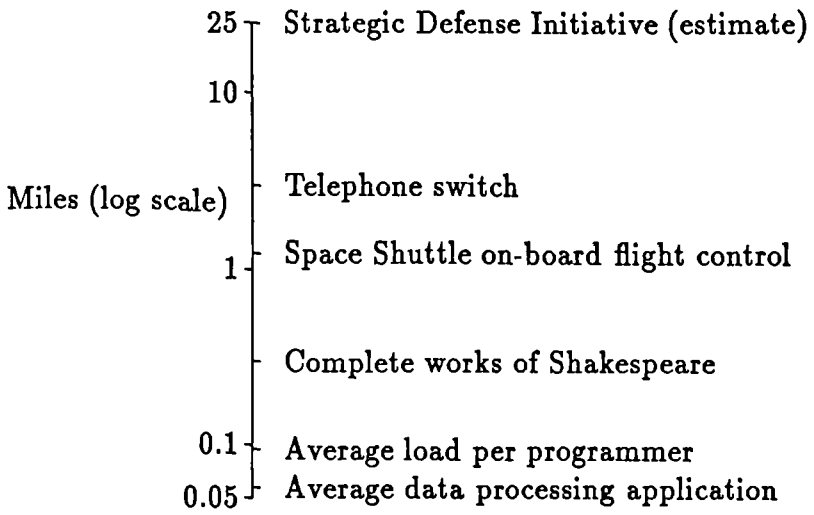

Figure 2.3: Examples of program size using Foster's metric.

Systems of about this baseline size were classified by Capers Jones as medium-sized systems [Jon77]. These system sizes and those still to be described are shown graphically in Figure 2.3.

Some indication of the difficulty of dealing with programs larger than a tenth of a mile is given by an example from outside the world of software. If we measure the complete works of Shakespeare [Cra62] on a similar basis, we find there are about 120,000 lines (0.3 miles). There are people who can justly claim familiarity across such a large body of text, but they are few and they may be expected to have devoted much more time to the study than is available to the normal programmer.

Systems larger than this are correspondingly more difficult to maintain, both because of the sheer volume of code and because their applications tend to be somewhat exotic. If we look around the 1 mile figure, a typical example would be the on-board flight control system for the Space Shuttle (actually about

500,000 lines, or 1.2 miles). Capers Jones classifies this as a "large" system, and the fact that it is also in the life-critical class makes it essential to adopt a very formal maintenance strategy with strong emphasis on process quality and improvement. That its maintainers have risen to the challenge is evidenced by the fact that the shuttle has never encountered a serious error in its flight control software, and by the estimate [Kel92] that this is the world's only maintenance project to have achieved the top level of 5 on the ratings scale of the SEI Model. (It can only be an estimate since the formal SEI Model does not directly address maintenance).

Between one and five miles is the range in which most telephone switching systems are found. Projects of this size are classified by Capers Jones as "super-large" — his highest category. These systems are not regarded as life-critical in the same sense as the Space Shuttle, but their reliability requirements are typically for not more than one major failure from any cause per 40 years. Again, it is not the size alone that makes them interesting as objects of maintenance.

If there is a practical limit to software size, it may at present lie at and above about 20 miles (8,000,000 lines). Inventories of programs can and do exceed this figure, but these are aggregations and not single systems. The possibility of achievement of this sort of size was publicly debated in 1983, when the now-cancelled Strategic Defense Initiative project was first mooted. The original estimate was for 10,000,000 lines (25 miles) of control software [Fle83].

2.5 Maintenance Statistics

Figure 2.1 showed the wide range of estimates of how much of software expenditure goes on maintenance, and until recently the accepted consensus has been one of high and rising costs. As an absolute amount of money, maintenance costs are indeed high; but the evidence supports neither the assumption that it accounts for the great majority of software costs, nor the assumption that the proportion is rising over time.

In a paper by the author in 1991, these assumptions were examined and rejected, at least for data processing applications [Fos91]. The paper argues that maintenance represents a constant percentage of software costs, at between 50 and 55%, and suggests that managers achieve the constancy by controlling the rate at which old programs are phased out and replaced by newly developed ones. An implication of this is that cost-effectiveness may not yet be a widely applied factor in such decisions.

The paper goes on to suggest that, if program lifetime is indeed being controlled to keep maintenance costs a constant proportion of expenditure, then overall improvements in the maintenance process will be measured by extensions in average program life and not by changes in overall expenditure. This is unfortunate, because program lifetime has been little studied [Ken90]. The general surveys that have been carried out have generally been patterned on [LS80], which contained no representative questions on the age of the software portfolio. (Program age did figure in part 2 of their questionnaire, but only to record the age of a sample application which was itself not randomly chosen). How-

ever, more recent work in Japan shows that the issue is now beginning to be addressed [TT92].

2.6 Improving the Maintenance Process

There have been numerous attempts over the years to "tackle the maintenance problem" with a variety of weapons, which can be classified broadly into two main types: improvements to the development process in the hope that better designed programs will require less maintenance; and direct attempts to improve the maintenance process itself. Early developments in high level languages and in the adoption of the structured programming discipline are these days taken for granted as better practices than their predecessors, and there have been many attempts to capitalise further by expanding the ideas.

So-called fourth generation languages (4GLs) have been one such attempt. The name covers a variety of development tools (being more a marketing than a scientific term), but it is generally applied to application generators that help designers to solve problems within more or less restricted domains. The expectation was that by promoting extremely rapid development times, maintenance would all but cease to exist, being replaced by further redevelopment as soon as change was required. That initial promise is now seen as somewhat flawed, though the reasons are not purely technical. 4GLs are by their nature designed for fairly specific application domains, which means that a wide variety of them have been produced. The process of standardisation, so much a feature of the more successful third generation languages, was notable by its absence, and with

reduced market share for each offering the tendency was for each 4GL to remain the property of its sole manufacturer. Users who were perfectly happy with the performance of their chosen 4GLs now find themselves vulnerable to their suppliers being taken over or otherwise going out of business; and if that does not happen, they then find themselves locked into using whatever range of machines their 4GL supplier decides to support. Not a few former 4GL converts are now in the process of migrating their software back to COBOL (e.g. [McD92]).

Formal methods have long been regarded as a great future hope for software, with their promise of rigorous mathematical proofs of correctness of programs. Claims have been made that programs that have been proven correct will require no maintenance; but these claims fail to take into account that corrective maintenance accounts anyway for only a minority of the effort. Formal methods have not yet advanced to the stage where they can be used on large-scale projects, and there has been little study of how incremental change will later be imposed on the mathematical proof structures.

Particularly in recent years, object-oriented programming has achieved considerable acceptance in software development, and the flexibility of its attributes such as inheritance and dynamic binding have been promoted as being automatically better for maintainers too. These claims have been regarded by many as self-evident; yet studies have called the benefits into question, since by their nature they can put program analysis and comprehension beyond the range of present-day tools and methods [WH91, LMR91].

Of direct assaults on the maintenance process, great efforts (and progress) have been made in the areas of reverse engineering and re-engineering [CC90].

Reverse engineering is the first stage of re-engineering, and it refers to the process of analysis of the source code in order to (re)discover underlying structures and promote understanding of the operation of the existing code. Gaining such understanding has been reported to take large amounts of programmer time: a survey of 800 programmers by the Mellon Bank found that they spent 29% of their time simply studying the documentation and code [Til87]. Other such estimates have been 50–90% of maintenance time [Sta84], and 30–60% [DBSB91]. The broader process of re-engineering starts with code analysis, but has the further goal of using the discovered structure as a base for reconstruction of the program, such that the structure is more cleanly reflected for the benefit of subsequent maintenance operations. The state of the art in both these subjects is well reflected in [vZ93].

2.7 Maintenance as Investment

Running through many of the published papers on software maintenance is a familiar litany. It runs something like: "Software maintenance is little understood, yet large amounts of money are spent upon it. Any new understanding must be good, and within such large expenditure even small technical advances must be worth many times their cost."

This is admittedly something of a caricature, yet a maintenance manager contemplating investment in a new maintenance tool, or in training costs for a new method, is likely to have little more available by way of justification. The problem here is that justifying expenditure on software within the normal

business is itself a little understood process. It is conventional to appraise investments according to their costs and expected future income, but the future income of a program which supports a business without directly generating revenue is difficult to calculate and currently the subject of research [Cle91, WDK93].

For managers in other fields, the ideal way to justify expenditure is to put forward an assessment of the financial implications of a proposed change in an activity. The calculation will include both costs and benefits, and will compare the net benefit with the costs involved. The result will be expressed in terms of a return on investment, which will also enable competing demands on available money to be ranked against each other. (Should we buy this new maintenance tool, or should we spend the money on additional advertising instead?) We suggest that software maintenance is more amenable to this kind of calculation than is software development, and the model put forward in the thesis will make its calculations on that basis.

2.8 Chapter Summary

Software maintenance is the process of making incremental changes to a software product in order to retain and enhance its value in the light of changing demand. Historically, it has suffered from negative attitudes towards it. However, recent years have seen changes in those attitudes and not the least of the effects has been a rapid rise in the number of research publications related to it. Maintenance accounts for just over 50% of all software expenditure, which

realisation has no doubt acted as a stimulus in that direction.

The difficulties of maintenance are not the same for all projects, as can be seen from a comparison of project sizes. The great majority of programs are relatively small, but extremely large ones do exist and these tend to have critical reliability requirements by the nature of their application.

The expense of software maintenance in an organisation can be considerable, yet maintenance actions are not subjected to the same investment analysis as decisions in other areas (including software development). Maintenance projects are typically controlled via fixed budgets, and the calculation of return on investment does not take place.

Chapter 3

Software Maintenance Models

3.1 Introduction

This chapter examines various models that have been put forward to explain and support the maintenance process. The classification takes an evolutionary view, beginning with the earliest models and progressing to the more recent (and broader) offerings.

In Section 3.2 we review a type of model that has been put forward from the early 1970s. It concentrates on the so-called modification cycle, which represents the sequence of actions taken by maintainers as they work through from change request to new release. Many variations exist, but at heart the models are prescriptive sequences of operations to be performed.

These models largely held sway until the mid-80s, when an expansion of viewpoints is apparent. The modification cycle is still central: but now there is new emphasis on other entities in the process such as types of knowledge and

forms of documentation appropriate at each stage. Section 3.3 considers these developments.

The above models all concentrate on maintenance as practised by maintainers. The next set to be presented broadens the perspective by considering maintenance as a continuously managed process, thus introducing the management aspects. The modification cycle is still understood to comprise the basic set of activities being managed, but the new generation of continuous process models looks for common elements across longer time periods, with emphasis on measurements of activity and effectiveness. These models are discussed in Section 3.4, and include models which take into account externally imposed constraints such as the SEI model of software maturity.

Along with the evolution of models has come a gradually increasing awareness of the importance of costs and benefit measurement, alongside the technical aspects of maintenance. Section 3.5 considers possible future developments along that path, and some of the obstacles to their creation.

3.2 Modification Cycle Models

These models present the maintenance process from the viewpoint of the individual maintenance programmer. They describe the sequence of actions taken in response to a change request, the sequence being known as the modification cycle [EM82]. The main models of this type up to 1983 are summarised in [CB86].

An early example is given in [Liu76]. Just three steps are enumerated:

understanding the problem; designing new program logic; and incorporating the revised logic into the program. Testing is treated as an activity external to the model, though of no less importance for that.

Later models extend the modification cycle to include the testing phase, and expand to varying degrees the number of stages. Many also include at the start a stage of problem analysis: thus in [Sha77] we have: problem verification; problem diagnosis; reprogramming (including rebuild); and baseline verification and validation. Similarly in [McC81] the four stages are: program understanding; identification of objective and approach for the modification (including detailed design); implementation; and revalidation. The second and the last of these stages are then broken down into respectively four and five sub-stages.

These and other modification cycle models differ in detail and in exactly where they consider the modification cycle to start and stop, yet their similarities are stronger than their differences. All have in common their view of maintenance as a sequence of stages, with only incidental reference to the objects being maintained and the information required for the process.

3.3 Entity Models

The summary of modification cycle models referred to above [CB86] was intended as more than a simple review. It used the commonalities between the models to derive a complementary list of the items of information required by the maintainers. These included information on the original requirements and specification, the architectural and low-level design and so on. These, along

with the source code, are the entities manipulated by the maintenance process.

Later models take this idea and expand on it, and in [HQ92] there appears a process description which starts with a modification cycle but goes on to define actors and their tasks, and documents and their information flows. Actors include review boards as well as maintenance team members; documents are such as the software problem report and software change proposal.

There is even more expansion in [CC92]. This paper, which along with [CB86] has Collofello as a co-author, presents maintenance as a sequence of tasks (stages), which manipulate entities (documents) in accordance with pieces of knowledge (of the information in documents; of other information for which there may be no formal record; and of how to perform tasks). Their model contains 17 tasks, 13 document entities and 29 knowledge items.

The IEEE Computer Society have for some years been studying the various models available, with a view to issuing a standard in the area. That has now been done, and IEEE Std 1219-1993 contains a seven-stage modification cycle augmented with the relevant document entities. The description of each stage includes the list of documents required, the detailed processing steps, the control actions (effectively a checklist to ensure correct processing), and the list of documents modified and generated. The stages begin with examination of the change request, and end with delivery and installation.

3.4 Process Improvement Models

The models so far described concentrate on how to perform maintenance. A more recent type of model takes a wider and more abstract view, seeking to identify the opportunities for improvement in the way the process is carried out.

Boehm's Spiral Model [Boe86, Boe88] is an early example of this abstraction process. It emphasises the setting of objectives and the evaluation of risks, and generally views maintenance more from the viewpoint of the manager than the maintainer.

Process improvement implies process measurement, and one such measurement system is described in [RU89]. The paper gives a model of a NASA maintenance process, and describes how a goal oriented approach was used to define a measurement programme, whose aim in turn was to identify parts of the process that were candidates for improvement.

The general theme of process quality came into prominence with the work of the Software Engineering Institute (SEI) in establishing guidelines for the evaluation of quality in software suppliers [Hum89]. The result of this work is the SEI Model, which defines five levels of process quality and contains questionnaires for carrying out the assessment process itself.

Despite its influence generally, the SEI model directly addresses only software development. This is, from the maintainer's point of view, a deficiency to be rectified. Work at the University of Durham in the UK has led to the development of a specific model for maintenance process improvement, in which the

steps of analysis, design of measurement, and modification of the process are described [HB92]. Simultaneously, the SEI Model itself is being reviewed by some of its users, who are proposing extensions for the assessment of maintenance operations [Dre92].

3.5 Cost/Benefit Models

Most maintenance managers would suggest that the costs of the software maintenance process are only too apparent, and to their managers in turn a primary objective is likely to be cost reduction.

Modification cycle models and entity models lend themselves to cost reduction exercises, because in breaking down the maintenance process into smaller stages they allow cost measurements to reveal the most expensive or resource-hungry tasks. These then become candidates for closer inspection.

The drawback of this approach, however, is precisely in its focus on costs alone. There is clear advantage in streamlining a task if the delivered quality of the product can be preserved, but it is impossible to identify those process changes which might actually increase maintenance costs while still delivering a higher net value in the end.

The previously cited method in [CC92] is for the most part cost-based in this manner, but it introduces a benefit-based element in suggesting that a causal analysis of errors could reveal those tasks whose improvement might yield a high return on investment. The suggestion is not expanded upon, and there is no indication as to how the return on investment might be calculated.

In a true cost/benefit model as we would define it, the net benefits of maintenance operations would be measured (or predicted) as the effects on the organisation's profits as a result of the changes and expressed in monetary units or more generally as a return on investment. One such model is the subject of this thesis.

3.6 Chapter Summary

Early models of the software maintenance process concentrated on the sequence of steps followed by the maintenance programmer. Model development has since expanded on this view, though many still retain the sequence as a component.

Process improvement models seek to evaluate the overall quality of the process and thereby identify areas for attention and change. They attempt quantification, but financial control is not necessarily a strong feature.

Cost/benefit models, such as the one presented in this thesis, are not yet prominent. They seek to establish the financial consequences of maintenance actions and plans, in terms of overall profit to the organisation. Benefit analysis has been established as a little-understood subject in the development world, but for maintenance the problem may be more tractable.

Chapter 4

The 7-Level Model

4.1 Introduction

This chapter introduces the author's 7-level model of software maintenance. The purpose of the model is to span a wider range than previously existing models, by considering technical actions and entities at its lower levels and investment factors at the higher ones.

Section 4.2 explains the broad outline of the model and introduces the seven levels. Sections 4.3 to 4.9 then describe the individual levels in more detail.

Section 4.5 gives the overall operation of a maintenance team. It goes into greater detail than the other sections, as this level is used extensively in the later chapters.

Overall, the 7-level model contains more detail than is strictly needed for the thesis. Many of the levels are therefore described only briefly, but the intention is to show that the model would provide a suitable framework for extensions to

the present work.

4.2 The 7-Level Model

The subject of software maintenance is large and complex. As with any such subject, it is useful to have a simplified model, to act as a focus for discussion. Without such a model:

- there is no standard framework of discourse within which different researchers and maintenance teams can discuss matters of common interest
- there is no standardisation of concepts, and time is wasted establishing and re-establishing exactly what is meant
- it is difficult to compare different organisations whose approaches may on the surface seem radically different
- it is difficult to isolate individual topics for discussion, since the boundaries are not clearly drawn.

Such a model is of most help if real maintenance organisations can be seen to map onto it, and should not be judged by the proportion of startlingly new ideas it presents. This model has been produced with that in mind. It may be seen as an extension of the existing models presented in Chapter 3.

The 7-level model breaks the subject area into levels in broad accordance with managerial levels in an organisation. The lowest levels are the most detailed; the higher ones take increasingly broad views of the process until at the

very highest, the software being maintained is seen as just one component of a company's asset base.

The levels of the model will first be listed from the highest down, then described from the bottom up. The list is:

Asset Level: software as a company asset

Portfolio Level: the set of software items owned and used by the company

Network Level: a subset of the portfolio (see detailed description)

Channel Level: the support chain for one software product

Team Level: a maintenance team

Function Level: a function performed within the team (e.g. request evaluation)

Topic Level: a component of a function (e.g. benefit assessment)

4.3 The Topic Level

Topics are the most detailed elements of the model. At the topic level, individual actions performed by members of the maintenance team are considered. Issues include:

- Allocation of time between tasks
- Reverse engineering techniques

- Methods and tools to support tasks

The individual steps of the modification cycle models presented in Chapter 3 would in the 7-level model be considered as topics.

4.4 The Function Level

Functions may be considered as groupings of topics, and the following list of functions is representative:

- Report Interface: the process by which customers of the maintenance team (who may be end consumers or other support teams) make their queries and problems known to the team
- Front Desk: the duty within the team of receiving, classifying and responding to reports
- Request Store Management: dealing with the backlog of change requests that must be scheduled for action (as opposed to having instant answers available)
- Escalation Procedure: passing on to other teams those reports that cannot be dealt with locally
- Change Design: converting a report into a proposed change
- Change Store Management: monitoring the set of designed changes that have yet to be delivered as a new build of the system

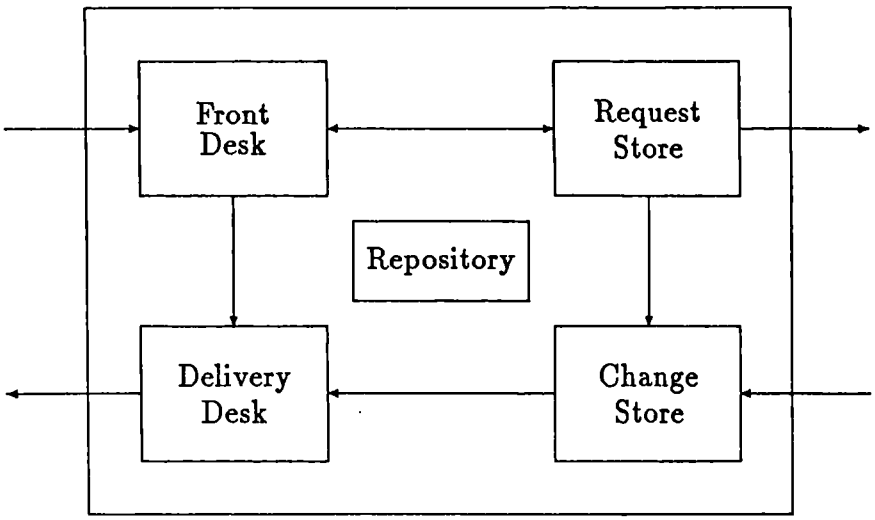

Figure 4.1: Maintenance Team Diagram

- System Build: entering the changes, and rebuilding and retesting the system
- Repository Management: dealing with the set of known solutions to problems, which includes new releases of the system as they become available
- Release Process: delivering new versions of the system

4.5 The Team Level

The team level, as its name implies, considers the work of a single maintenance team. It brings together the functions and topics for that team, and shows how they relate and interact.

Figure 4.1 shows the diagram that summarises the actions of the team. In the figure, the large rectangle represents the organisational boundary between

the team and its customers. What lies inside this boundary is directly under the control of the team; what lies outside it is not.

Outside the rectangle and to the left are the customers of the team. From them are received queries, problem reports and change requests.

Within the team, these communications are received by the **Front Desk** duty, which retains records of them. It is the responsibility of the Front Desk to provide answers/solutions, either directly or by passing on the request to a more specialised duty within the team.

The Front Desk has access to a set of known answers/solutions, which is represented in the model as the **Repository**. This represents information in a variety of forms: versions and variants of the software product(s), paper records of answers to frequently asked questions, etc. Knowledge that resides only in the heads of those performing the Front Desk duty is also regarded as being in this store.

If the solution is in the store or can quickly be generated from information that is, then it is immediately to hand and can be issued back to the customer without further delay. The arrow from the Front Desk to the **Delivery Desk** and thence out to the customer shows this flow.

Otherwise, a new solution must be generated. We have just accounted for the cases where this can be done quickly, so we now assume that the request must be queued until effort becomes available.

This queue is represented as the **Request Store**, which contains the backlog of unactioned requests. Customers would like this store to be empty at all times, but economics and staff availability usually dictate otherwise.

The management of the Request Store is an important function. Priorities must be assigned among its contents, and preliminary investigations and impact analysis performed in order to plan future work. The world changes and moves on, so these assessments must be revisited from time to time to keep them current [Fos89].

These preliminary investigations may reveal that the team does not itself have the resources or capability to provide the answer to the problem. For instance, a fault may actually lie in a software module provided (and maintained) by some other team or company. In this and other such cases, a request must be made to that team or company for a solution to be provided, and this team will then be the customer as far as they are concerned. On the diagram, such further teams lie to the right of the boundary rectangle.

If the team can deal with the request, however, then it will be dealt with as one of a repeated series of actions in which the highest priority request is taken from the store and (usually) a software change designed.

The holding store for these changes is the **Change Store**, into which solutions received from other teams are also placed.

From time to time, the decision will be taken to build a new release of the software, incorporating all new changes available. (There are exceptions to that "all": emergency action may require just a single vital change to skip past the rest).

New releases are then lodged in the Repository, from which they are available for distribution to the original customers.

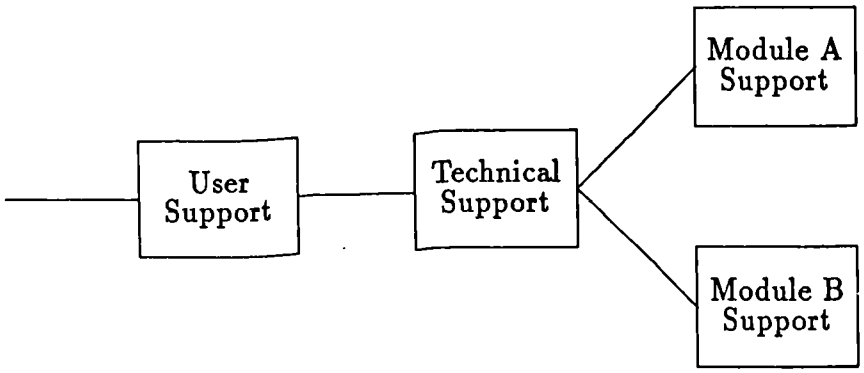

Figure 4.2: A Maintenance Channel

4.6 The Channel Level

The team model of figure 4.1 works well at that level, but it is rare for the whole support of a product to be carried out within a single team.

Where more than one team is involved, there will typically be a front-line user support team, which may call on a technical system support team, which may in turn call on separate teams for the different components (modules) of the product as a whole.

This situation is modelled by representing each team as a box after figure 4.1, and showing the boxes joined to indicate the chain of customer/supplier relationships that then exists.

The result is a maintenance channel diagram as shown in Figure 4.2.

Each of the team and channel diagrams is essentially a flow diagram for requests and their solutions, but the channel diagram takes the higher level view. (In a real situation, the channel diagram will typically be more complicated than

this example shows).

At the channel level, we are concerned with studies of the overall response to requests. For example, user satisfaction with product maintenance is greatly affected by response time between original request and installed solution. The channel of teams must itself be studied, as well as the response times of the individual teams. Communications between teams are also studied at this level: requests from one team do not reach the next instantaneously. If the boundary between two teams is also a boundary between two companies, contractual negotiations may increase the delays. Users are concerned only with overall responses, but the channel management must be aware of the details of the channel performance if they are to concentrate efforts where they will do most good.

4.7 The Network Level

Most maintenance teams have responsibility for several products, and the competing demands of these products on the team's resources cause interactions between the products.

The network level of the model is concerned with these interactions, and its basis is an augmented version of the channel diagram.

In this augmented version, separate channel diagrams are drawn for each product supported by the team, and these are then overlaid to show the broader network which is the set of teams and customers with which this team will interact and whose operations it will affect. The flow and queueing metrics

are then applied to the network, and bottlenecks are identified. As a result, channels may be reorganised and/or resources redistributed between teams.

Another version of the network level considers all products used by a particular customer or group of customers, and draws the augmented network of teams as it affects the customer. This permits the evenness of support across the product range to be examined and if necessary optimised.

4.8 The Portfolio Level

At this level, the concern is with the sets of products that support particular business functions of the company. It is valuable to assess and compare:

- the extent to which each business function is supported by software products
- any opportunities for extra products, or enhancements to existing ones
- the software investments and paybacks for the individual products

The benefit of this exercise (which is known as Portfolio Management and can be obtained as a commercial service) is that the evenness of support across business functions can be assessed and compared, and effort directed to where it will best serve the company as a whole.

4.9 The Asset Level

This is the highest level of the model. It considers the entire set of software products owned by the company, and is concerned with the total value of this investment and the overall costs and paybacks associated with it.

The benefit is that software can then be assessed in terms of its importance to the business as a whole, and ranked accordingly for strategic planning purposes. These include research and development allocations, as well as broad budget decisions.

A question which occurs here, is whether (and if so how) software assets should be represented on the books of the company, alongside its buildings and other visible assets. Some companies have indeed done this, but for most the immediate question is how to derive the appropriate figures rather than what the accountants should do with them next.

4.10 Chapter Summary

A comprehensive model of software maintenance has been presented, in which seven levels of description combine to link the most detailed actions with their financial consequences in terms of investment.

The most important level as far as this thesis is concerned is the team level, which encapsulates the local concerns of a single maintenance team. At this level can be seen the flows of change requests from submission to resolution. Measurements of flow, queue lengths and delays are possible at this level, and

can be related to the team level diagram.

Chapter 5

Formal Description of Model

5.1 Introduction

The model that lies at the centre of this thesis is presented in detail in this chapter. It is a part of the previously sketched 7-level model, selecting those attributes and functions that are of concern to an individual maintenance team and its immediate management.

The model must, if it is to be successful, assist the maintenance manager who has control of the rest of the team and its own operations, but who must direct those operations in response to outside pressures over which the team can exert little or no control. A related problem faced by the team is that of imperfect information: customers and suppliers are rarely willing to impart all information that could be helpful (usually for commercial rather than technical reasons) and the model must either avoid requiring information which in practice would not be available or have an explicit mechanism for dealing with uncertainty.

The description will first outline the business and numerical assumptions that underlie the model, and which are in effect the criteria that it must meet. The team diagram is then reintroduced in Section 5.4, and the model's components are derived from it. It is shown that the form and use of the team diagram viewpoint permit the collection of metrics that are of value to both the customer and the team itself, and differences between the customer and team viewpoints of these metrics are discussed.

The descriptions of the model's parameters and calculations then form the bulk of the chapter. Each quantity is defined as either a description or an equation; where an equation is not possible, an indication of the algorithm that performs the calculation is given.

The calculation as a whole is necessarily iterative in nature, and the mathematics of continuous behaviour cannot be applied. Section 5.11 discusses the consequences.

Section 5.12 sets out the limitations of the model which might have to be addressed in any practical implementation.

5.1.1 Conventions

The model to be presented in this chapter takes various input parameters and performs calculations on them to generate intermediate and output results. These quantities are given by definition, calculation and/or equation (as appropriate) in the text, and definitions, calculations and equations share the same numbering scheme.

For convenience, two tables are provided in which all the parameters and results are listed, together with their symbols and the references of their definitions or the equations in which they are defined. Table 5.1 lists the parameters, and Table 5.2 lists the intermediate and output results.

Each of the quantities used in the model has both a symbolic and a textual representation. Textual representations appear in italic type to stress their special meaning, in this and subsequent chapters. There is at the end of the thesis an index of the terms used, so that the uses of each may be seen. Index entries are triggered either by the use of the textual representation or by the appearance of the symbol for the quantity concerned.

5.2 Business Assumptions of Model

Since it is a requirement (Section 1.2.4) that the model be realistic in accepting basic facts of commercial life, it is necessary first to document the necessary assumptions about those basic facts. There are three:

1. The ultimate driving force behind decisions is money. In general, anything a business does translates (or should) eventually into profit, and it is the pursuit of maximum profit for minimal expenditure that keeps a business solvent.

   Stated in such bald terms, this could be the classic description of any get-rich-quick, cowboy operation. But profit is a complex, long-term quantity. Companies which are to be successful in the long term need to spend money on maintaining their reputation, which may include the finance of

Parameter	Symbol	Defined
acceptance cost	C_A	5.31 p. 77
acceptance time	τ_A	5.15 p. 65
base build cost	C_B	5.6 p. 59
base build time	τ_B	5.7 p. 59
change delay	T_C	5.29 p. 76
change effort	E_C	5.1 p. 54
distribution rate	U	5.17 p. 65
external change cost	C_X	5.2 p. 54
frequency base	H	5.23 p. 71
incident frequency	R	5.22 p. 70
incident value	S	5.20 p. 68
initial installations	I_{init}	5.10 p. 64
new installations	NI	5.11 p. 64
retired installations	RI	5.12 p. 64
staff	F	5.5 p. 57
staff cost	C_S	5.3 p. 54
time max	T_{max}	5.9 p. 63
time now	T_{now}	5.8 p. 63
unit upgrade cost	C_U	5.32 p. 78

Table 5.1: Parameters of the model

Result	Symbol	Defined
available change effort	E	5.28 p. 76
change list	CL	5.30 p. 76
distribution cost	C_D	5.33 p. 78
distribution start time	T_D	5.16 p. 65
frequency base count	Q	5.21 p. 70
frequency multiplier	G	5.24 p. 71
gross change benefit	B_G	5.25 p. 71
incident type count	J	5.19 p. 68
installations	I	5.13 p. 64
net change benefit	B_N	5.26 p. 71
optimum release time	T_{opt}	5.37 p. 79
priority	P	5.27 p. 73
release benefit	B_R	5.36 p. 78
release completion time	T_{DONE}	5.18 p. 65
release cost	C_R	5.34 p. 78
release time	T_R	5.14 p. 64
release value	V	5.35 p. 78
total change cost	C_T	5.4 p. 55

Table 5.2: Intermediate and output results of the model

activities that in themselves make no profit at all; nevertheless, in the long term view, profit is still the underlying motive.

2. There are exceptions to the first assumption! Certain actions are driven by legal or safety requirements, and a company wishing to stay in business at all must take those actions. While it would be possible in theory to assign penalties as negative profit and weigh the risk of breaking the law, it is an unwise company that does so. There is explicit provision in the model for the treatment of these mandatory issues when they arise as software change requirements.

3. In an imperfect world, assumptions replace actual data but are best documented so that the impact of new knowledge can later be assessed. The model will support this by allowing actual data to be used where available, but accepting all inputs as parameters that can be documented, and changed later as may be necessary. In the special case of assumed benefits, figures are accepted in a form which as far as possible isolates the effects of individual assumptions from one another and makes changes as painless as possible.

5.3 Numerical Assumptions of Model

In addition to the recognition of commercial aspects, the model is required to provide quantified predictions for use in decision making. It must therefore deal in numbers and measurements, and have a coherent structure so that different

quantities are related and can take part in calculations. Specifically:

1. The model will deal with a form of token as a fundamental unit. This token is created by a customer (i.e. outside the team) as a change request, and ultimately returned to the customer as a solution to the request or in response to a decision by the customer to withdraw the request. The return may, however, be delayed for an arbitrary length of time if the request finds itself placed at the bottom of a long priority list. Thus: within the team, tokens are neither created nor destroyed, though they may be stored.

2. Tokens will have various attributes, expressed as data attached to them. These attributes will include intrinsic worth (benefit of supplying a solution), and history information such as times of transit through the various phases of the model, and associated costs. For reference, Table 5.3 lists these attributes.

3. Phases of the model may also have data associated with them, such as the average time to design a change.

4. It must be possible to extract summary data, such as overall costs of the team's operation, from the individual data for passing on to higher management.

5. The model must be able to use detailed data from lower layers of the 7-layer model. If for example a new reverse engineering tool is under evaluation, it should be possible to feed in the estimated gain in efficiency

Attribute	Symbol	Attribute	Symbol
change delay	T_C	*historic cost*	
change effort	E_C	*identity*	
customer charge		*importance*	
description		*incident frequency*	R
event times		*incident type*	
external change cost	C_X	*net change benefit*	B_N
frequency base	H	*priority*	P
frequency multiplier	G	*total change cost*	C_T
gross change benefit	B_G		

Table 5.3: List of Token Attributes

on one part of the process and predict the future effect on the entire operation.

5.4 Team Diagram Revisited

Figure 5.1 repeats the team diagram, drawn according to convention with customer(s) on the left and supplier(s) on the right. A general description of its operation has been given in Chapter 4. We will now concentrate on the quantifiable aspects of its use.

Incoming tokens arrive via (1). Each has attributes of *identity* (a unique reference for the token); *description* (text associated with the token which will at least include the customer's description of the problem); and *importance* (an

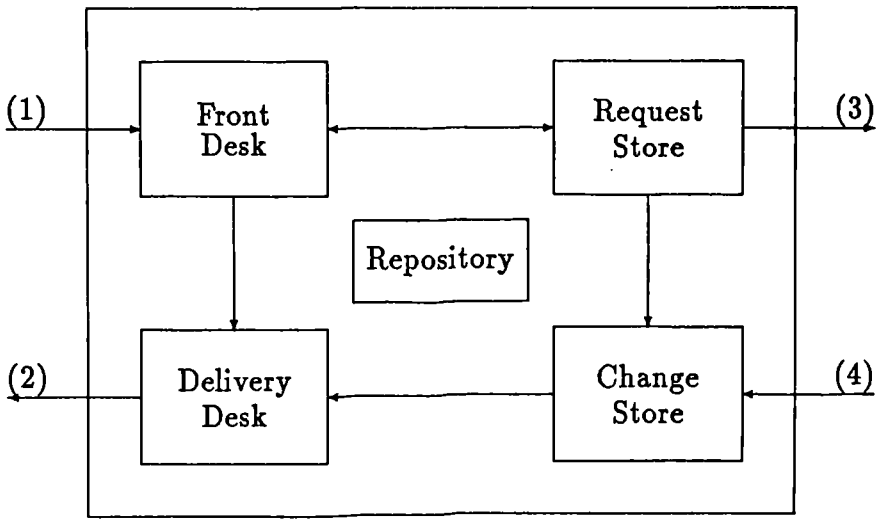

Figure 5.1: The Team Diagram

attribute which the team would like to quantify, but which will be delivered in a form decided by the customer alone). The *importance* represents the customer's initial view of the priority of this token, and is distinguished from the subsequently calculated *priority*, which takes into account both benefits and costs.

There is also considered to be a *historic cost* attribute (see below), which at the time of receipt is zero.

The token's *description* attribute is qualitative rather than quantitative. It is assumed that it will be augmented by the team to record actions, opinions and results as the token is processed; but it is not required or used by the model, which deals only with quantified data. The *description* attribute will therefore be ignored for the rest of this formal discussion.

The *historic cost* attribute, in contrast, is wholly quantitative. As the token is further processed, the costs of so doing will be added to the attribute, which

may be kept either as a simple running total or in the manner of an itemised bill. In further descriptions, this accumulation of value will generally be taken for granted. The *historic cost* is money that has already been spent, and as such it takes no part in future projections and plans.

A further attribute is known as the *event times* attribute. Whenever a token passes from one functional area of the team diagram to another, the time of the transition is recorded in this attribute (which therefore has the form of a list of entries). Later, metrics information may be extracted from this list.

Tokens are received by the Front Desk, and there marked with the time of receipt in the *event times* attribute. Access to the Repository will show whether a solution is immediately available; this will happen if the token is a simple request for information, or if the solution is to provide a more recent release of the software which is already available. In this case, the token passes directly from the Front Desk to the Delivery Desk, and delivery takes place. At delivery, the token is returned to the customer via (2), and its attributes, augmented with the time of delivery, are recorded in the Repository.

No Immediate Solution If no immediate solution was available, the token must next pass from the Front Desk to the Request Store, where it will await attention. At this point, it gains the further attribute of *priority*, which relates to but is distinct from the *importance* that was originally present. The *priority* attribute will be developed further in Section 5.8, but for now it is sufficient to observe that *priority* will be a scalar quantity which primarily expresses the team's view of the cost-effectiveness of delivering any particular token in the

queue. It will also have a secondary function that if a token's *priority* lies below a given threshold level, the further processing will have been deemed to cost more than it is worth, so unless events cause the *priority* to be revised upwards, such tokens should never leave the queue unless withdrawn by the customer.

There will also be cases in which the calculation of *priority* becomes irrelevant, because a change is mandated by legislation or other over-riding force. In such cases, there is a mechanism to bypass the normal mechanism and assign an effectively infinite *priority* to a change. (See Section 5.8.2).

Although *priority* is a scalar quantity, it is not directly an input to the model but instead the result of a calculation involving costs and benefits. The costs are the estimated costs and delays of completing the design of the change, and are expressed as the *change effort* and the *external change cost.*

Definition 5.1 *The* change effort, *denoted by E_C, is the estimated future effort that must be expended by the team in order to bring the token into the Change Store.*

Definition 5.2 *The* external change cost, *denoted by C_X, is the estimated future direct expenditure that is required in order to bring the token into the Change Store. The* external change cost *does not include the cost of the* change effort.

These components express the expected costs of getting the design as far as the Change Store. We will next introduce two terms which will allow groupings of the components, for which we will need first to define *staff cost.*

Definition 5.3 *The* staff cost, *denoted by C_S, is the amount of expenditure associated with one unit of staff effort. It is expected that the actual figure*

will be derived in accordance with normal company standards — usually as a combination of direct salary costs plus assigned overheads.

Definition 5.4 *The* total change cost, *denoted by C_T, is the total future expenditure associated with carrying out the change to which it refers. It is the sum of the costs of effort and external expenditure, as given in the following equation.*

$$C_T = C_X + E_C C_S$$

Discussion of benefits follows a similar but more complex mechanism, and will be discussed in Section 5.6.

Management of the Request Store is the next process to affect and be affected by the token and its attributes. Management of the store (as far as the model is concerned) consists of a periodic re-examination of all the tokens in the store, with a re-evaluation and updating of their attributes.

The Design of Change Any token at or near the top of the priority queue is then a candidate for the fix process, which in the familiar view means that a maintainer takes a pending request and designs a software change that will satisfy the request, placing the completed design into the Change Store for later incorporation into a new release. The model actually takes a somewhat different view of this process, though it is to be stressed that the difference does not affect the way that the maintainer works.

The problem is that while in principle the maintainer may take a request and carry it through to the design of a change in one smooth action, in practice it doesn't always happen that way. Jobs get interrupted by others of higher

urgency; a diagnosis may reach a point where it is discovered that the change must be made in a module that is owned by another maintenance team; it may suddenly be discovered that an error report actually arose from misoperation of the system rather than from anything requiring a software change; etc etc. For all these reasons, it is impractical to have the model assume a smooth, one-way flow of tokens along the line from the Request Store to the Change Store.

Instead, the model takes a view in which tokens are considered to remain in the Request Store until the resolution (withdrawal or design of change) is complete. In this view, the actions of the maintainer add to the information associated with the token, and correspondingly add to the *historic cost* attribute and (hopefully) steadily decrease the future *total change cost.* On completion of the design, the information will be sufficient to specify the change exactly for inclusion in a build, and also to specify the unit test to be applied after the build to check that the particular change works. At the same time, the future estimate of the *total change cost* will become zero, and only at that point is the token transferred from the Request Store to the Change Store. The steady reduction in the *total change cost* will cause the *priority* to rise correspondingly, which reflects the intuitive notion that a design once started should probably be continued, but still allows it to be interrupted if another token should gain an even higher *priority* at any stage.

It is in the design of change that the main people resource limitation occurs, because all activity that contributes directly towards the next release is concentrated here until the build starts. The number of staff available to perform this activity will be important to the model, and when later it refers to *staff* it will

be according to this definition:

Definition 5.5 *The effective number of people available at any time t for the design of change is known as the* staff *level and is denoted by $F(t)$. The future projection of variations in this value is referred to as the* staff profile. *The units in which* staff *are counted must be compatible with the units of effort, cost and time, so that numerically, s staff working for time t will deliver st units of effort at a cost of stC_S.*

External Change Design Designs that reach the Change Store through direct action of the team are not the only possibility. As has been remarked, some changes will be in externally maintained modules, and this is where the team must in its turn become a customer. The identification of a token as being of this type will take place while the token is in the Request Store, and will cause a request to be generated to the appropriate supplier via (3) in Figure 5.1. The eventual arrival of the solution from the supplier via (4) will cause the transfer of the token from the Request Store to the Change Store.

Token Creation Revisited There is an important conceptual point associated with this part of the process, concerning the rule that the team neither creates nor destroys tokens. It has been explained that the customer does create and destroy tokens, and suddenly we have the team acting in turn as a customer — an apparent inconsistency.

We have not so far discussed in any detail the communication channels between customer and team, or between team and suppliers. They have been

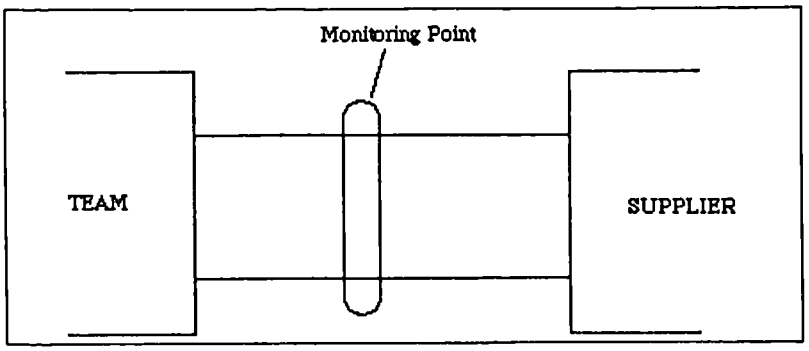

Figure 5.2: Channel and Monitoring Point

assumed to be mere pipes or conduits. However, it will be seen that some important measurements are made in those channels, and it is for that reason convenient to give them some structure of their own. Figure 5.2 shows a channel between a team and a supplier, though it could equally have been the channel between customer and team. A Monitoring Point is shown on the channel, which for the purpose of the model is assumed to be an entity placed mid-way along the channel, and which observes the passage of requests and solutions, collecting performance statistics by examining their attributes. The statistics collection will be discussed further in Section 5.5 but here we may note that the Monitoring Point may be regarded as the source and sink of tokens, creating them in response to requests from the left and destroying them as solutions pass back to the left.

Beyond the Change Store As far as the Change Store, tokens move individually from one point to another. From here on, they move in batches as sets of changes are incorporated into new releases. Between the Change Store and the Delivery Desk, the processes of editing the source code, recompilation

and testing take place. The exact details will vary from project to project, and are no concern of the model, which summarises the whole process in two parameters: the *base build cost* and the *base build time.*

Definition 5.6 *The* base build cost *of a release, denoted by C_B, is the combined cost of editing the changes into the source code, rebuilding the software, performing system tests upon it, and all the other costs that are entailed in producing a working product. The* base build cost *is assumed to be a constant, and is given to the model as a parameter.*

Definition 5.7 *The* base build time, *denoted by τ_B, is the elapsed time between starting a build and delivering the release to the customer, but excluding any extra elapsed time due to individual tests on the changes incorporated.*

When a build, including its associated testing, is complete, the tokens representing the individual changes incorporated in the release are moved collectively from the Change Store to the Delivery Desk. Here, a fresh attribute known as the *customer charge* is calculated and added to the token's existing attributes: this represents the amount of money the customer is required to pay for that change. (The calculation of charge is a matter for the commercial agreement between team and customer, and is not specified within the general model). Finally, all attributes are copied from the token and recorded in the Repository, and all attributes except the *identity* and *customer charge* are stripped from the token, which is then returned via (2).

5.5 Flows, Delays and Queues

The model's strict association of tokens with maintenance actions, its control over the creation and disposal of tokens, and its association of attributes directly with tokens, all make it possible to derive consistent metrics to describe the process as a whole. This section considers the aspects of this that are independent of costs.

First, recall that tokens are created only at a Monitoring Point, and circulate only within the team to the right of that Monitoring Point before being returned to the Monitoring Point for destruction. Without any need to be aware of how the team or its suppliers operate, and with no need to receive data from them apart from the defined data associated with the token itself, the Monitoring Point can determine the rates of flow of requests and solutions, and note the distribution of delay times between requests and solutions. It is further aware at any time of how many tokens are in progress within the team, and from observation of the *importance* attached by the customer it can present these statistics broken down by that *importance.*

Within the team, the visibility of flows, delays and queues is far more detailed. For example, the number of tokens in the Request Store is exactly the difference between the inflow from the Front Desk and the combined outflow of withdrawals (back to the Front Desk) and change designs (to the Change Store). The main flow from the Request Store to the Change Store is constrained by available money and effort, and the overall change in performance from any particular improvement or extra resource allocation can be calculated in terms

of the resulting change to flow rates, and consequential changes to delays and queue lengths.

5.6 Incidents and Benefit Multipliers

We now turn to the consideration of the calculation of benefits. We begin with a fairly simple hypothetical example, and examine the calculation of net benefit from that point of view.

Suppose that the software product is installed at various different sites over some geographical area. Suppose further that a communications module within the software is able to signal to a central engineering point that the failure of a hardware component has been detected, and to request the attendance of an engineer to repair the fault. Finally, suppose that an error within this software is causing occasional false alarms, each of which entails a wasted trip by an engineer. How do we calculate the benefit of correcting this error? Several factors must be taken into account:

- The average cost of a wasted trip is clearly a factor, and we may reasonably assume that a good estimate could be provided.
- The number of wasted trips (say, per installation per year) is equally clearly a factor, and again we may assume it to be available.
- The number of installations, and how the number is expected to change in the future, will influence the benefit.

- The expected lifetime of the installations is also required: the longer it is, the greater will be the total benefit accrued.

- The estimated time of delivery of the solution to the installations themselves is a factor: the correction does not start earning money until it is in the field.

- The rate of distribution of upgrades may be significant: if each involves a site visit in itself, there will be limitations of available effort and other resources. The full benefit will not be available until the last installation has been upgraded.

- If the system has only recently entered service, it is likely that new installations are regularly being added. If new installations are assumed to be fitted at source with the latest available version of the software, upgrade distribution costs may be avoided for these installations and benefits realised more quickly.

Generalising, we will say that each wasted trip in the example is one *incident* of that particular type. The benefit of making the change will then be the average cost of one such incident, multiplied by the total number of such incidents that would have occurred (within the *time max* of the model) had the change not been made.

To bring some order to this list of factors, we now introduce Figure 5.3, which depicts many of them in graphical form.

The X and Y axes of the graph are respectively elapsed time (likely to extend

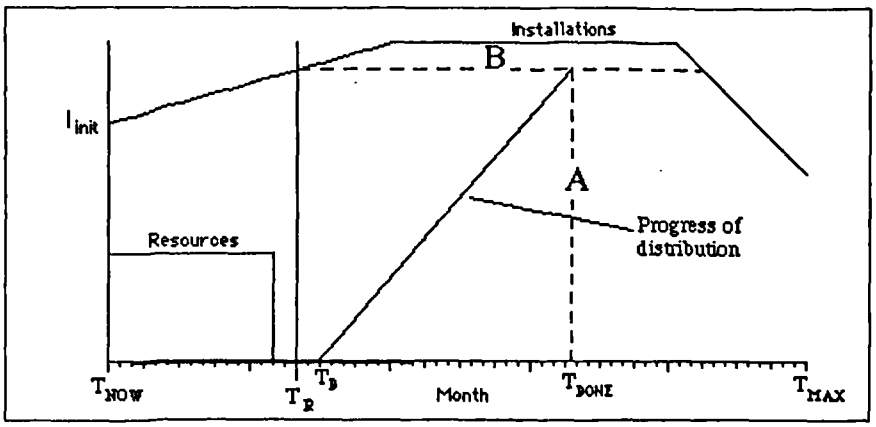

Figure 5.3: Release and Distribution Graph

over years) and number of installations in the field. The origin of the X axis is "time now" or T_{now}. For consistency, a standard time period of five years, with the month as the basic time unit, will be assumed in all illustrations and examples.

Definition 5.8 *The origin of time as far as the model is concerned is the present moment, known as* time now *and denoted by* T_{now}.

The main solid line shows the projected number of installations, rising from I_{init} now to an eventual peak, and with a following plateau after which installations are steadily removed from service until the graph ends at time T_{max}.

Definition 5.9 *The time horizon of the model, and the limit of iteration for all calculations, is known as* time max *and denoted by* T_{max}. *It may represent the time at which the last installation is projected to be taken out of service, or it may be chosen as some earlier time beyond which events are considered sufficiently distant to be ignored.*

(Note that the piecewise linear form of this graph is for explanation only; such simplification is not required by the model). The variation in the number of *installations* over time will be referred to as the *installation profile.* It is calculated from three input parameters as described in the following set of definitions.

Definition 5.10 *The number of* initial installations I_{init} *is the number of installations in the field at time* T_{now}.

Definition 5.11 *The profile of* new installations NI *is the time-varying function of those installed per time unit.*

Definition 5.12 *The profile of* retired installations RI *is the time-varying function of those permanently removed from service each time unit.*

Definition 5.13 *The number of* installations *in the field at any time* T *is denoted by* $I(T)$. *The future projection of variations in this value is known as the* installation profile. *It is calculated as:*

$$I(T) = I_{\text{init}} + \sum_{t=T_{\text{now}}}^{T} \text{NI(t)} - \sum_{t=T_{\text{now}}}^{T} \text{RI(t)}$$

Working to the right along the X axis, the time at which a solution is delivered to the customer is shown as T_R, with a subsequent period of customer acceptance assumed which culminates at time T_D. The customer acceptance period is taken to include any time necessary to prepare to begin the distribution of the release.

Definition 5.14 *The* release time T_R *is the time at which a new, tested version of the software is handed over from the maintenance team to the customer, and at which the customer begins the acceptance period.*

Definition 5.15 *The* acceptance time τ_A *is the period of time during which the customer is checking the acceptable operation of the new release and preparing for distribution, before commencing distribution itself.*

The model assumes that distribution of the software to the installations in the field begins immediately after the completion of the acceptance period. Therefore, if we denote the *distribution start time* by T_D, we have

$$T_D = T_R + \tau_A \tag{5.16}$$

In general, distribution of upgrades takes place via two routes.

Firstly, the upgrade is delivered to the factory or other source of new installations. From time T_D onwards, every new installation is assumed to contain the new software. Secondly, a programme of upgrading the $I(T_D)$ existing installations commences, achieving a *distribution rate* of some number of installations per unit time. The progress of the upgrade programme is shown as the "Progress of Distribution" line on the graph, and is complete when $I(T_D)$ upgrades have been made, or earlier if the retirement of installations will have reduced the number of upgrades to be made. These latter quantities are defined as follows:

Definition 5.17 *The* distribution rate U *is the number of installations per time unit, in which the new release will be installed.*

Definition 5.18 *The* release completion time T_{DONE} *is the time at which all installations in the field have been upgraded with the new release. It is calculated as:*

$$T_{\mathrm{DONE}} = T_D + \frac{I(T_D) - \sum_{t=T_D}^{T_{\mathrm{DONE}}} \mathrm{RI}}{\mathrm{U}}$$

In the above equation, T_{DONE} appears twice: once as the quantity to be calculated, and again as a component of the sum on the right hand side. There is no obvious way to eliminate it from the right hand side, if only because I and RI are non-analytic quantities. Because of that, though, any calculation of the formula must be carried out iteratively. The double appearance of the term then ceases to be a problem.

5.7 Benefits

In Section 5.6, the software change considered was a straightforward one, in which it was clear that the individual saving per incident was relatively easy to estimate in financial terms. Often, this is not the case. If we consider only faults in the software, they may manifest themselves as merely irritating (giving rise to customer complaints, perhaps), or even purely cosmetic, such as a spelling error in a report that is never seen by an end customer. Enhancements can be equally troublesome: how does one put an objective value on a change that simply improves the software's human interface, but that gives no other obvious benefit?

In defining a solution to this question, three guiding principles have been followed. They are:

1. It is a commonplace in business that where objective measurements are not available, subjective estimates must be and are substituted.

2. (Divide and conquer) Difficult questions should be split into smaller or-

thogonal components.

3. (Keep it simple) A solution that is just good enough, is good enough.

Here, the problem is to set a value for the saving per incident, where it is known that a number of factors may contribute. Suppose it is believed that a particular change will both reduce the number of customer complaints, and reduce the incidence of incorrect hardware fault diagnosis by the company's own engineers. The question to be answered is: "What level of expenditure in making this change would represent the borderline beyond which the change should not be made?" This is a difficult question in itself, which will be all the worse if it has to be re-evaluated in the future in response to a decision by the company to, say, increase perceived quality by making greater efforts to deal with complaints effectively (meaning that it's then prepared to spend more money per complaint).

However, the question can be broken down into four orthogonal sub-questions:

1. How many customer complaints are being made due to this problem, per installation per year?

2. How much would the company be prepared to spend in the reduction of customer complaints, per complaint avoided?

3. How many hardware diagnosis errors are being made, per installation per year?

4. How much would the company be prepared to spend in the reduction of hardware diagnosis errors, per error avoided?

At first sight, this may not look like much of an advance. One subjective question has been replaced by two questions which are far more objective (questions 1 and 3), but there remain two questions (2 and 4) which are subjective in just the same way as was the original.

However, an important gain has in fact been made. Questions 2 and 4 are no longer linked to the particular change; their answers will apply equally for all change requests which involve those factors. Their answers are now system-wide parameters, and the information associated with the individual change request is now far more tractable.

The number of different factors that are identified in this way is known as the *incident type count*:

Definition 5.19 *The* incident type count J *is the number of incident types that have been identified as important in the calculation of the benefits of changes.*

In general, if a total of J different factors have been identified that affect the particular system, we require the setting up of a system-wide vector $S_{1...J}$, and a similar vector $R_{1...J}$ for each change request token. Entries in R represent observed frequencies of events, while the unit benefits of reducing those frequencies are contained once only in S.

Definition 5.20 *The* incident value *vector S contains one entry for each basic incident type, giving the estimated value of reducing by one the number of incidents of that type that occur. One such vector is global to the model, and its values are provided as parameters.*

We must now examine more closely Question 1 in the list. It asks for the frequency of the incident "per installation per year", but this form of denominator will not serve in all cases. As examples we may suggest two other possibilities: some proposed changes may have no effect on normal operation, but may reduce the costs associated with each new installation. (Looking ahead to Chapter 8, change request 6 on page 108 is of that type). Alternatively, a change may carry benefit over the remaining years of the system's life, but be of such a nature that the change benefit will be proportional only to the number of years itself.

We now have three examples in the "per what?" category, and more may be added if they are found necessary in particular cases. We will next describe the calculations for those three, which we will call the incident *frequency bases*. Referring back to Figure 5.3 on page 63:

- For the number of years left ("per year"), the benefit starts at T_D and lasts until $T_{\max}$. Its value is simply

$$T_{\max} - T_D$$

  (Clearly, another example could have been based on the time at which distribution is complete).

- For the number of new installations, the base is just the number still to come after the *distribution start time* T_D. This is

$$\sum_{t=T_D}^{T_{\max}} \mathrm{NI(t)}$$

- For the "per installation per year" case, the calculation involves the area under the *installations* graph, but only that part of it that is to the right

of the line of distribution progress. It is split into two smaller areas. One, labelled A in Figure 5.3, represents installations that have been upgraded in the field; the other (labelled B), represents those installations that have been provided new after time T_D. Two factors make the actual calculation quite complex:

- The representation on the figure implies (area B) that the last new installations are also the first to go when numbers start too decline. We actually calculate on a last-in, last-out assumption.
- Variations in the shape of the *installation profile* may change the geometry of the situation by varying where lines intersect. This too must be taken into account.

The actual algorithm for "per installation per year" will not be presented here.

Our three examples of *frequency bases* need not be restricted to that number; in general:

Definition 5.21 *The* frequency base count Q *is the number of* frequency bases *that have been identified as important in the calculation of the benefits of changes.*

For each incident type, we must then specify both the frequency with which it occurs and the base for that frequency.

Definition 5.22 *The* incident frequency *vector R for a particular change contains one entry for each incident type, giving the estimated difference in fre-*

quency of that type of incident that the change will bring about. One such vector is associated with each change token.

Definition 5.23 *The* frequency base *vector for a particular change contains one entry for each incident type, giving the index in* $1 \ldots Q$ *of the frequency base associated with the corresponding* incident frequency *entry in R.*

For any given *release time*, we can now define the *frequency multiplier* vector, which will apply across all changes.

Definition 5.24 *The* frequency multiplier *vector G has one entry per* frequency base. *For a given* release time T_R, *its entries contain the actual values by which the* incident frequency *values in R must be multiplied, to give the number of incidents saved by including a particular change in a release.*

5.7.1 Gross and Net Change Benefits

We are now in a position to define the *gross change benefit* B_G:

Definition 5.25 *The* gross change benefit, *denoted by* B_G, *is the total assumed value of the change once it has entered field service. It is taken across all installations and until the* time max $T_{\max}$ *of the system.*

$$B_G = \left(\sum_{j=1}^{J} S_j R_j G_{H_j} \right)$$

Next, to derive the *net change benefit*, we need to know the future cost associated with the change. But this is just $C_T = C_X + E_C C_S$, so the *net change benefit* is represented by

$$B_N = B_G - C_T \tag{5.26}$$

We also observe that we have defined a rejection criterion for changes: if $B_G < C_T$ then the *net change benefit* B_N will be negative, and the change should not be attempted.

5.8 The Definition of Priority

The planning of software releases is a juggling act involving benefits, costs and resources. We have now introduced each of these elements into the model, and must next consider how to specify the optimum release. We wish to maximise the net benefit, within the constraints of available resources.

Constraints might in general apply to many kinds of resource, including direct expenditure (cash flow), available staff effort, and such things as time allocation of specialised facilities. We do not attempt to treat all possibilities in the model; instead we focus on the specific and almost universally observed restriction of available effort (Section 2.2 on page 10).

Our problem, then, is to maximise the net benefit within the constraint of available staff effort. In this section, we discuss how the concept of *priority* contributes to the process.

5.8.1 Normal Priority

The *priority* of a request for change (a token) represents the team's view of its relative value, compared with all others in the queue. In terms of the finance-driven view of their motivations given on page 46, that translates directly into consideration of which tokens will deliver maximum benefit for minimum ex-

penditure.

If all resources (including effort and money) were freely available, we would not need to discuss *priority* at all. We would simply include all changes that had net positive benefit in the next release, and with no resource limitations we would be able to issue that release immediately.

In practice, our limiting resource is human effort: we must use that effort to best advantage. If two units of effort are available, we use them better for two fixes at one effort unit each that each benefit us by £5000 than by choosing an alternative fix that takes the two effort units for a net benefit of £9000.

The *priority*, then, must express the desirability of a change in terms of its net benefit relative to the proportion of the scarce resource that it consumes.

But the benefit and resource expenditure have now been derived, in terms of the parameters of the model and the attributes of the tokens. In fact, the *priority* of a token is now seen to be simply

$$P = \frac{B_N}{E_C} \tag{5.27}$$

In passing, we may note that the definition of *priority* also reveals another expression of the rejection criterion mentioned on page 72. These are the ones whose *net change benefit* is negative, and by examination of the above equation we see that these are correspondingly indicated by a negative *priority*.

5.8.2 Infinite Priority

As has already been indicated, there are situations where the financial calculation of *priority* is unnecessary, because other factors make it essential that

a particular change is carried out with utmost urgency. These situations include changes mandated by legislation, and in safety-critical systems where the absence of change introduces a safety hazard.

Rather than try to force the model to treat these changes as imperative by choosing some large but finite benefit to accredit to them, we actually go further by allowing their *priority* to be set to an infinite value directly. Infinity can be a dangerous concept to introduce in a model, but here the only subsequent operation in which it will take part is the sorting of the tokens in the Request Store, and an infinite value then floats naturally to the top.

When more than one token has infinite priority, they will all float to the top of the list but the model defines no ordering among them. This is not a problem: the point is that they must all be selected for the next release, and if they cannot be (i.e. if not enough effort is available) then the release is simply unacceptable as a whole.

5.9 Release Value

The value of a release, as we require it, is generally to be expressed as the return on investment, taking into account all the costs and benefits of that investment. Return on investment can be calculated in many different ways [WDK93], but for the purpose of this thesis a simple calculation will be used. We will take the value as being the difference between the benefits and the costs within the time horizon of the model, and we will ignore any future discounting of the value of money. We observe that in a practical implementation, this formula should

be replaced by one that reflects the accounting practices of the organisation involved.

Since we can assume here that we are dealing with a specific projected release time, the calculation of *priority* for each token can be undertaken, and in the course of that calculation we will discover the net benefit $B_N(c)$ associated with any individual token c. We require two further results:

- A list CL of which tokens will actually be included in the release, since only their net benefits will contribute to the *release value.*
- An assessment of the cost C_R of the release itself.

Each of these turns out to be fairly straightforward.

5.9.1 Change List

For the list of tokens, we examine those in the Request Store, selecting them in order of *priority* (highest first) until we cannot select any more within the resources available up to the time of release. (But the model will never extend the selection to include tokens with negative *priority*). This presupposes, of course, that we have calculated the available resource.

For that, we may need to distinguish between resource types. People's time is the most obvious, but in an exotic application the use of special equipment or the time of one specific expert may be at the highest premium. Here, and in order not to complicate the situation beyond bounds, we will assume that the available effort of the maintenance team is the limiting factor.

The available effort, then, will be the product of the number of maintainers available and the time to the start of the build — not the time of release. In practice, the number of maintainers can be expected to fluctuate over time, and we may wish to explore the effect of adding people to the team during the course of preparation of the release. We recall the concept of the *staff profile*, which is the time-varying projected number $F(t)$ of maintenance staff available, and we note that the total *available change effort* or $E(T_R)$ is

$$E(T_R) = \sum_{t=T_{now}}^{T_R - \tau_B} F(t) \tag{5.28}$$

where τ_B is the *base build time.*

This is not yet quite the whole story, for some changes will be processed by teams other than the one under consideration. These changes incur direct cost as their *external change cost*, but they are also subject to time limits. With each such change, we assign a non-zero value to a new parameter: the *change delay.*

Definition 5.29 *The* change delay, *known as* T_C, *is the time before which this change cannot be available for inclusion in a release. Thus, no change can be included for which*

$$T_C > T_R - \tau_B$$

Given E and for each change T_C, the selection of tokens for the release proceeds as follows:

Definition 5.30 *The* change list CL *is derived by the following steps:*

1. Arrange the tokens in the Request Store in descending order of priority. *Begin with the* change list *empty.*

2. Starting at the head of the list of tokens, accept each token into the change list *if and only if it has positive priority, and its acceptance will not cause the sum of* change effort*s* E_C *for the tokens in the* change list *to exceed the* available change effort E.

3. Also exclude any token for which $T_C > T_R - \tau_B$.

As a by-product of this process, any token that is excluded but has infinite priority (see Section 5.8.2) should be reported, since in such a case the release cannot fulfill a basic requirement placed upon it.

5.9.2 Release Cost

There remains to be calculated the cost associated with the release itself, known as the *release cost* or C_R. This is the sum of:

1. The *base build cost* or C_B
2. The *acceptance cost* or C_A
3. The *distribution cost* given by $C_U I(T_R + \tau_A)$ (where C_U is the *unit upgrade cost* or the cost of upgrading one installation) and known since the release time is a postulate.

Definition 5.31 *The* acceptance cost, *denoted by* C_A, *is the total cost incurred by the customer between receiving a new release of the software and beginning*

its distribution. It thus includes, not only the cost of any customer testing, but also any costs incurred in the preparation for release.

Definition 5.32 *The* unit upgrade cost, *denoted by C_U, is the average cost of upgrading one installation in the field to the new release of software.*

Definition 5.33 *The* distribution cost, *denoted by C_D, is the total cost of upgrading all the installations that are in the field at the time of release. Hence:*

$$C_D = C_U I(T_D)$$

Thus we have the definition of the full *release cost* C_R:

$$C_R = C_B + C_A + C_D \tag{5.34}$$

From all of that, we are then able to express the *release value* at *release time* T_R as

$$V(T_R) = \left(\sum_{c \text{ in CL}} B_N(c, T_R) \right) - C_R \tag{5.35}$$

where the expression within the right hand side also gives the *release benefit* B_R

$$B_R(T_R) = \left(\sum_{c \text{ in CL}} B_N(c, T_R) \right) \tag{5.36}$$

5.10 Optimum Release Time

We are now in a position to define the calculation of *optimum release time* T_{opt}. If we are prepared to iterate to a solution (and we are), the calculation of *optimum release time* T_{opt} is simply a matter of calculating *release values* $V(t)$

for a sufficient range of possible release times t, and then selecting the time which offers the highest value. So at last:

$$V(T_{\text{opt}}) = \max(V(t)) : T_{\text{now}} \leq t \leq T_{\text{max}} \tag{5.37}$$

Discussion: There are two possible cases in which use of the result of this calculation may need care:

1. The value $V(t)$ may never rise above zero: that is, $V(T_{\text{opt}})$ may be negative or zero. In this case, the model has determined that the most cost-effective course of action is not to make any new release of the software at all.

2. There may not be a single unique maximum. Perhaps there are two or more equal peaks, or even a plateau at the peak. In this case, the model is reporting a choice between equally valid release times.

5.11 Discontinuities

There are questions that might be asked of an implementation of the model, whose answers could be generated automatically if the model possessed simple continuous formulae linking its inputs to its outputs. Continuity would hold out at least the hope that the following operations could be carried out algebraically instead of by trial and error:

Tolerancing: The formulae could be differentiated with respect to each input variable in turn, leading quickly to estimates of the sensitivity of output values to small changes in the values of the inputs, and isolation of which

input parameters were most sensitive and should therefore be scrutinised most carefully

Finding Local Maxima: Available tradeoffs could be investigated, if they themselves could be expressed as formulae. (Example: suppose the product of distribution rate and cost per upgrade turns out to be constant: which pair of values is best?)

Without continuity, the model's outputs will from time to time jump suddenly from one value to another, even in response to arbitrarily small changes in the input values.

Unfortunately, continuity is not a property of the model. There are three reasons.

Firstly, the operation of the model involves a step at which the change requests are sorted into priority order. If two or more priorities are nearly equal, then very small variations in input values can switch the outcome of the sort to a new state.

Secondly, after the sort a cutoff is imposed on the change list according to the effort available. A change is either included or not; arbitrarily small input variations will cause step variations in the content of the list of accepted changes.

Thirdly, the selection of which release is best rests on locating the maximum point on an irregular curve which may have several local maxima at similar heights. Again, arbitrarily small input variations can cause a step variation when a new local maximum becomes the global maximum.

This is unsatisfying, but it arises from the situation being modelled and not from any local limitation of the model itself.

5.12 Limitations

There is one glaring omission in the model, which would in general need to be rectified in practice. It is that the arrival of change requests is assumed to stop after the plans are drawn up: there is no element of prediction for new requests that will arrive before the release is made. An accurate prediction would require a crystal ball beyond our present reach, but knowledge of the past history of arrivals of requests would allow an assumption of future arrivals based on the past rates, costs and benefits. Actual change requests will of course be incorporated into the model as they arrive to see the effect on the plans, but prediction could reveal early the need to build in any required contingency.

There are some simplifications in the model, where complete generality has been sacrificed for the sake of clarity. An example is in the expression of the *distribution rate* U, which is given as a constant but which could more generally have been made a time-varying function. Equally, the point in time at which new installations are assumed to carry the new release is assumed to be the same as the start of the field upgrade programme. In such cases, the extra calculations would not be severe. The judgement was, however, that the approximations were likely to be more acceptable in practice than the more complicated data entry.

The model assumes that the *base build cost*, *base build time*, *acceptance cost*

and *acceptance time* for a release are all constant values. The system test and acceptance processes will, however, include some component for individual tests of the changes incorporated. If required, the costs and times of these individual tests can be added to the model as extra token attributes, to be incorporated respectively into the calculations of *change effort* and *external change cost.*

The model in the form given here calculates the *release value* as a simple profit value, ignoring both future discounts on the value of money and any assessment of the ratio between profit and investment. Most organisations will normally take both these factors into account when comparing investment opportunities, though exact methods do vary. In order to take into account the preferences of any particular organisation, it will be necessary to rework Equation 5.35 and (for the future value of money) its contributory equations 5.26 and 5.34.

5.13 Chapter Summary

The handling of change requests and the subsequent construction and distribution of a new release of software have been described quantitatively, together with calculations that permit the outcome of actions and decisions to be seen. Effort, time and money are the main quantities that appear in the model, and money its chief guiding concern.

The model's inputs are a set of change requests and their associated data, and a set of more general parameters that include descriptions of the build and distribution environments. The model predicts the financial and other

consequences of a decision to release the software earlier or later, and selects the release time which will yield the greatest net benefit.

The model may be used in this way to select the best release time, but it can also quantify the effects of process improvements by altering the values of the parameters that would be affected and seeing the results in terms of the new best plan.

Small changes in the parameters fed to the model may result in large differences in the figures generated. This is an inevitable consequence of the existence of the sorting process by priority among the change requests.

Chapter 6

Exploring the Model

6.1 Introduction

This chapter describes the use of the model in preparation for the chapters on its implementation. Section 6.2 introduces the concept of the baseline plan, which is the result of running the model with best-estimate values. Section 6.3 covers the basic exploration and evaluation of this plan, and Section 6.4 discusses the evaluation of alternative situations and their comparison with the baseline.

Section 6.5 looks at how to cope with uncertainties in input values, and how to tell which are the most sensitive and should therefore be checked most carefully.

Section 6.6 then discusses the formation of alternative plans, based on events that may occur and whose consequences need to be anticipated.

Section 6.7 contains remarks on the possible dominance of certain quantities in the determination of results.

6.2 The Baseline Plan

The baseline plan corresponds to the best estimate of future events, and will be the central planning document. Its construction is carried out with the aid of the model and is quite simple in concept: one establishes best-estimate values for all the input parameters of Table 5.1, and one performs the calculations described in Chapter 5. The requirements of Section 5.10 (Equation 5.37) mean that many of the calculations must be iterated many times.

The primary results of the calculations are the values of the *optimum release time* T_{opt} and the *release value* $V(T_{\text{opt}})$ at that point. The secondary results are then the *change list* CL and the *release completion time* T_{DONE}.

In the absence of other constraints, these results will be accepted as the release to aim for. However, if there are other constraints such as a limit imposed on the *release completion time* then the baseline release will be selected as that which delivers the best value within the constraints.

6.3 Exploring the Baseline Plan

The baseline calculations yield one optimum scenario, but all scenarios had to be calculated in order to select it. By examination of these sub-optimal alternatives, it will be possible to see and to comment on whether the baseline plan is clearly the best, or whether any other possibilities are nearly as good. If they are, then the decision is not clear-cut and the fact needs to be reported.

6.4 Alternative Plans

Once the baseline plan has been established, it becomes possible to see the effects of possible changes in the parameter values.

It may be proposed, for example, that a new member be added to the team in order to reduce a backlog of change requests. This is a simple change in the model's *staff* parameter, in which its value increases by one after some allowance of time for recruitment and perhaps training. This in turn increases the *available change effort*, which increases the number of changes accepted into the *change list*. The extra *net change benefit* from the added changes then adds to the total *release value*, and from the amount of that increase an assessment can be made of how well justified the extra team member would in fact be.

Probably most alternative plans may be dealt with in a similar manner, by adjusting the appropriate parameters and observing the difference in outcome of the calculations. Some, however, will also involve added constraints on the solution, such as a deadline for the full delivery of the new release to the field. A simple deadline will invalidate any *release time* that cannot meet it, and reduce the range of the search for the *optimum release time*. In the most general case, an arbitrary constraint would invalidate some particular subset of the possible *release time*s, and the search for the optimum would proceed with the rest.

6.5 Stability

Each plan accepts all its parameters as though they were known with perfect accuracy, but of course they are not: they are merely best estimates.

It is therefore prudent to ask whether possible inaccuracies in the estimates would affect the outcome of the plan, and if so to what extent. This is a process well known to designers of electronic circuits, where component values cannot be exactly known and the effects of tolerances must be taken into account.

It is here that the non-linearities inherent in the model have their greatest effect, and they do make the process rather difficult. If, for example, we decide that the value estimated for a particular parameter may be out by 10% either way, we may re-run the model with those boundary values and observe that (say) the plan remains intact. But perhaps a 5% increase in value would have led to a radical change in the plan through crossing one area of non-linearity, only to find the original plan reinstated at 7%.

We will not offer a perfect solution to this problem, though we do note it as a subject for future research. Here, we need observe only that if extreme values do not suggest a change in the plan, the presence of an anomalous region in the unexplored part of the range need not upset us: it would represent an alternative plan which itself could not be relied upon since by its definition the parameter value is not considered reliable enough to guarantee hitting it.

The exploration of stability in practice, then, is a process of varying parameter values to the limits of their likely ranges and checking whether the resulting plan would differ significantly from the baseline. If it does, then the first prac-

tical reaction is likely to be to re-examine and improve the estimate in order to reduce its range; if this is not enough, the variation must be noted as a risk that the planned release date (or value, or content) may not be met.

6.6 Risk Evaluation

The accuracy of estimated values is one source of variation in the final outcome, but there is another which arises from radical changes in one or more parameters. This is the risk analysis part of the planning process, whose purpose is to test how the plan might be impacted by external events. Small impact means the events would be manageable if they occurred; large impact may imply the need to prepare additional contingency plans.

The specification of what events (risks) should be checked for is a matter of practical judgement, and no model can substitute for that. But the effects of those events are seen as changes in the parameters that are provided to the model, and the impacts on the plan are seen as the consequent changes in the output values. The model thus plays its part in the risk evaluation process.

Risk evaluation, then, is conducted by assessing the new parameter values should the event occur, and re-running the model to observe the effect. On the model, a risk is implemented just as an alternative plan.

6.7 Dominant Values

The exploration of variations in the input parameters may be expected to reveal that some are more important than others in the determination of the final result: that is, that the values of the less important parameters may vary quite widely without materially affecting the outcome. This is a beneficial effect, for it is what allows attention to be focused on the major parameters in terms of checking and reassessing their likely values.

An example of dominant values will be seen in Chapter 8, where in the list of eight change requests to be studied, four of them generally come out to have much higher priorities than the other four. This usually guarantees their presence at the head of the priority table and in the recommended release.

In general, those changes whose benefits are given in terms "per installation per year" are likely to gain the highest priorities. That is because the *frequency multiplier* for these is derived from the area under the *installations* graph and is typically much larger than the alternative multipliers. It seems that an area measure of that sort is not intuitively estimated by people, which can also make the figures generated by the model larger than might otherwise be expected.

6.8 Chapter Summary

At its most basic, the model may be used to predict the outcome of a given maintenance situation. However, its main value is in the ability to explore alternative situations. Input values will be in large part estimates, and by

varying their values it is possible to see how uncertainties will affect the main prediction. From that, values whose estimates may need further refinement may be identified.

Future events (risks) that may upset the plan may also be explored, giving an opportunity for contingency plans to be created as and where necessary.

In most situations, there will probably be some parameters whose effect on the outcome is stronger than that of others. Change requests whose impact will be felt in proportion to the number of installations and to the number of years of use are likely to be associated with this effect.

Chapter 7

Implementation of the Model

7.1 Introduction

This chapter describes the implementation of the model as a computer program. Section 7.2 describes the development package used, and Section 7.3 describes the implementation in more detail, with full descriptions of the display screens that are seen by its user. Section 7.4 notes some limitations of the implementation.

7.2 Development Package

HyperCard™ is a development package which runs on the Apple Macintosh™ computer, and like the machine itself is well suited to applications with a graphical interface. Applications developed in this way do not stand alone but run under the main HyperCard program. This acts analogously to a conventional

language interpreter, though there is a certain level of automatic precompilation.

User applications are referred to as "stacks" because they appear in the manner of a stack of index cards, of which any one can be at the front and therefore visible at any one time. In terms of a more conventional description, cards are alternative screen displays, which may be presented to the user in any desired order and in response to user actions.

Cards may contain two main types of active element: buttons and fields. Buttons are sensitive to mouse clicks from the user, which trigger scripts (program fragments) written as part of the application. Fields may also be sensitive to clicks, but their primary purpose is as holders of fixed or editable text. Artwork may also be drawn on cards.

The scripting language of HyperCard is comprehensive and flexible, but can be slow in heavily iterative applications such as the present model. To get around this, program fragments may also be coded in a conventional high level language (C and Pascal are supported) and included in the application in their object code form. Use has been made of this facility in the implementation.

7.3 The Implementation

The model is implemented as one main HyperCard stack and several subsidiary ones. The subsidiary stacks are a stack called "Background Index" to be described first, and an expandable set of "situation stacks" which are cloned from the main one.

The Background Index stack is important to the implementation, but does

not implement any central part of the model. As the model runs, cards are created and destroyed as scenarios are calculated or change requests are added and deleted. HyperCard has no built-in mechanism for keeping track of such changing collections of cards, but the model has to maintain lists of what exists and update the lists as changes occur. It is this process that is carried out by the handlers in the Background Index stack.

The situation stacks provide the means whereby alternative situations may be investigated in the model and compared afterwards. The mechanism is simple in concept. After initial data entry and the preparation of the baseline plan, the entire main stack (containing both data and programs) is copied and named. Each time another variant is created, it too is copied and named. The names of these new stacks are collected in a menu available to the user, and comparison of situations is then a matter of switching from one to another by selecting the menu items.

The mechanism of situation stacks is wasteful of space because of the copying of all the program information that takes place. Greater recourse to the "stacks in use" facility would ameliorate the problem, but it was considered acceptable in a prototype system.

7.3.1 The Main Stack

The main stack of the model is called "Priority Control" and contains all the meat of the programs and data. There are six backgrounds whose cards the user sees:

- The Welcome background, which is no more than an introductory display
- The Summary Screen background, which with its single card contains the summary graph of profit against release dates
- The Scenario background, each of whose cards describes the consequences of selecting one particular release date
- The General Parameters background, whose single card contains the general parameters of the model
- The Change Requests background, each of whose cards contains the data for one change request
- The Change List background, whose single card just acts as an index into the change requests. Clicking on the title of any request takes the user to that card.

The middle four of these will now be illustrated and described.

Summary Screen

Figure 7.1 shows a typical summary screen. At the top are two headers. The left header announces that this is indeed a summary screen display; the right that it refers to the baseline plan. When alternative plans are explored, this header is a reminder of which plan is currently on display.

In the centre is a graph, which shows the calculated *release value* V as a function of *release time* T_R. The graph starts out at a negative value, reflecting

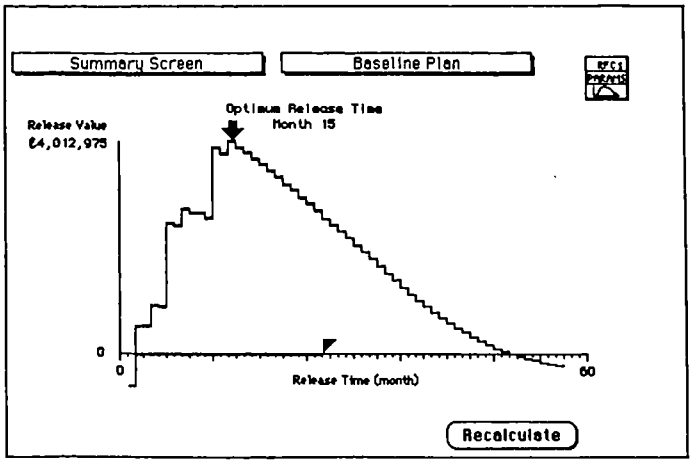

Figure 7.1: Sample Summary screen

the expense of a too-early release in which there has not been time to include any changes at all. It then rises by steps, as effort to include one change after another becomes available. Between steps it falls slowly; here extra effort is also becoming available, but it needs to accumulate over a month or two before another change can be included.

The graph eventually reaches a peak, which represents the optimum *release value* occurring at the *optimum release time* T_{opt}. An arrowhead marks the peak, with a legend that confirms the time at which it occurs.

After the peak (in this example) there is a steady decline as the benefit of inclusion of any further changes is outweighed by the loss in overall benefit due to the delay in delivery.

The vertical axis of the graph is automatically scaled to accommodate the peak value, which is shown against the axis. This figure must therefore always be checked when comparing different graphs, since two visually similar ones may in fact be drawn to very different scales.

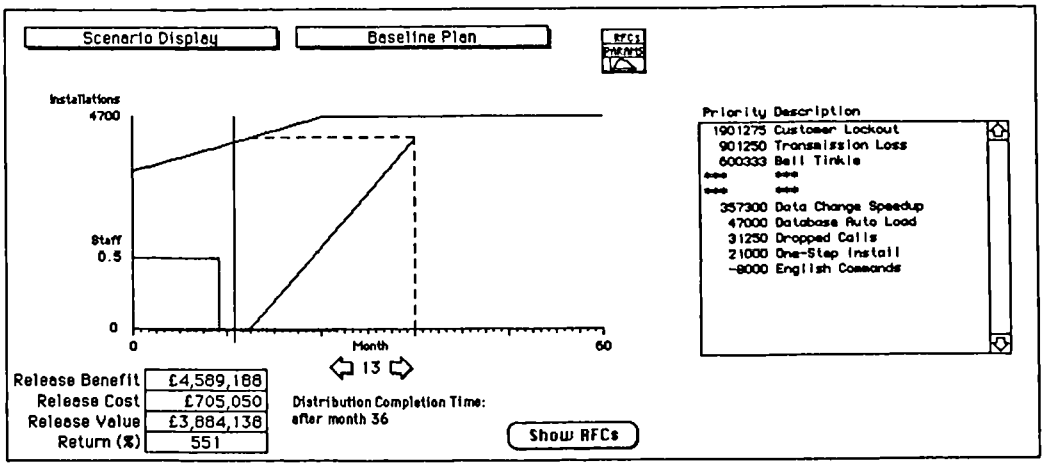

Figure 7.2: Sample Scenario Display

A small triangular marker sits on the time axis, at month 26 in this example. The user can drag it to any month; it sets a limit to the number of scenarios (see below) that will be generated in detail for subsequent examination. Scenario generation is time-consuming, and those well after the peak are of little interest.

At the top right is a square button divided horizontally into three parts. The user clicks on one of these parts to be taken to that section of the model.

Scenario Display

To each value of *release time* on the summary screen (up to the time set by the limit marker) there corresponds one scenario display screen which may be examined for further details of the implications of a release at that time. An example of a scenario display is given as Figure 7.2. It has the same two headers and its main feature is again a graph, though this time it is a composite one. In the example, the *release time* under consideration is after month 13, and a solid vertical line extends the whole height of the graph from this point on the

time axis, as a marker.

To the left of the release marker and nearest to the origin is another graph (here of constant height) showing as a function of time the available *staff F*. This graph is auto-scaled vertically so that it always occupies the same part of the overall picture, and its peak value is marked on the vertical axis. Its right boundary (solid line) occurs to the left of the release marker by an amount equal to the *base build time* τ_B, so the area under it is the *available change effort E*.

In some cases there will be a vertical dashed line just before the right boundary of this graph. When present, it indicates spare effort which is not enough to accommodate another change into the build.

Starting further up the vertical axis, the number of *installations I* is then plotted as a function of time. Here, it rises steadily over two years to a maximum of 4700 and then remains constant at that value. The graph is automatically scaled such that its peak uses the full vertical extent available on the display.

Using the same vertical scale, the progress of field updates after the release is then plotted. It is the diagonal line to the right of the release marker. Its start is delayed by the *acceptance time* τ_A (here two months) so it begins at the *distribution start time* T_D. It then rises at the *distribution rate.*

Also from the *distribution start time*, new installations are assumed already to contain the new release. The dashed line which starts on the *installations* graph at the *distribution start time* and runs horizontally to the right reflects this. When it intersects with the distribution progress line, distribution is complete. The completion is marked by a vertical dashed line from the meeting point down to the time axis, and a legend below the axis confirms the comple-

tion time.

To the right of the composite graph is a table, containing a list of all the outstanding change requests with their assigned priorities. The list is divided into two sections by a pair of rows of asterisks; above these are the changes to be included in the release and below are those rejected. There are two rows of asterisks because of one special case: if a mandatory change cannot be included in the release through insufficient effort being available, it will be listed between the rows. This is a means of flagging what would in practice be a severe headache for the maintenance manager. The table is shown to the right for convenience here, but in the actual implementation it has to overlay part of the composite graph in order to fit within the physical display screen. It is shown or hidden via the "Show RFCs" button below the composite graph.

Below the composite graph is a pair of arrow buttons (with the *release time* shown between them). These buttons will switch to the previous or next scenario, for comparison of release dates in the form of a slide show. There will sometimes be situations where the *optimum release time* as calculated will be unacceptable due to other constraints, and it is then necessary to explore other scenarios until the best within the constraints is found. The slide show is convenient at such times, and it may be interspersed with visits to the summary screen by clicking with the mouse anywhere within the main graph area. From the summary screen, any scenario may be visited directly by clicking at or above the appropriate point on the time axis, including on the line of the summary graph itself.

At the bottom left of the display are the primary cost and benefit figures for

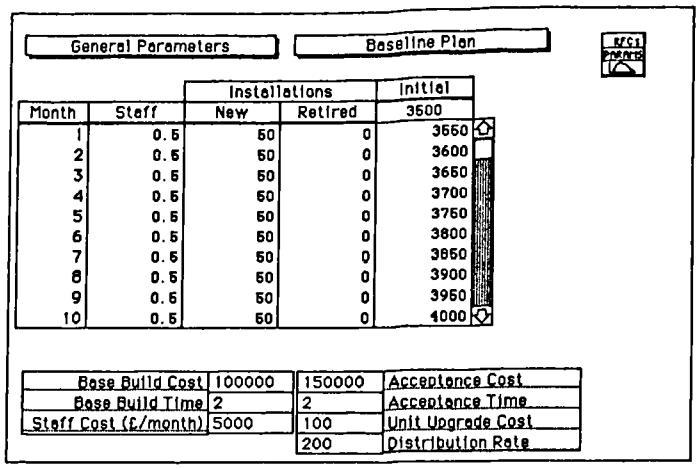

Figure 7.3: General Parameters screen

the scenario. The table shows the *release benefit*, the *release cost*, the *release value* and the *return on investment.*

General Parameters Screen

Figure 7.3 shows the data entry screen for the general parameters of the model. The multi-column table is a scrolling field which actually contains 60 lines — one per month within the time horizon (i.e. up to *time max*). One column gives the available *staff*, and to its right are two columns giving the numbers of *new installations* and *retired installations* for that month. The rightmost column then shows the calculated number of *installations* actually in service: just above it is a field giving the number of *initial installations.*

The other parameters are entered in the fields at the bottom of the screen, and are labelled according to the terms used in Chapter 5.

The buttons to the right of the scrolling field lead to dialogues which make it easier to set constant values across ranges of months.

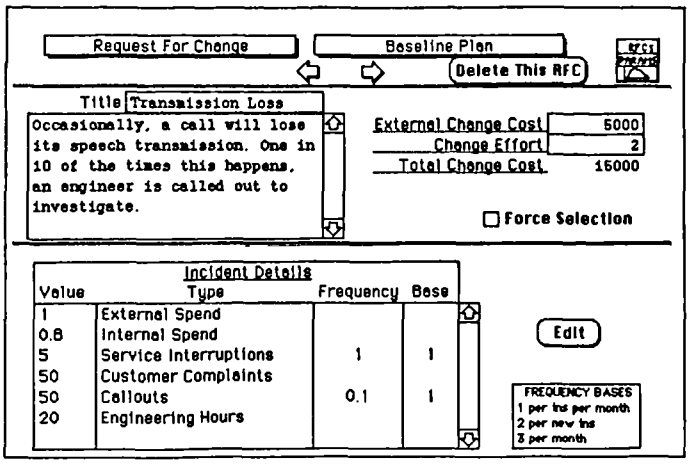

Figure 7.4: Data entry screen for a change request

Change Request Screen

Figure 7.4 shows the data entry screen for a change request. It is filled in with data for one of the example requests used in Chapter 8.

At the top, with the standard components of the display header, are two arrow buttons which permit sequential stepping through the other change requests and a further button for use when a request is to be deleted.

Below this and to the left are entered the short title of the request and a longer description; the description is for illustration but is not otherwise used by the implementation.

To the right of the description are the estimated cost and required effort for the request, and the total cost is calculated by reference to the *staff cost* parameter entered on the General Parameters screen.

Just below the cost information is the check box (here shown clear) which when checked will mark this request as mandatory.

The lower half of the screen shows the assessed benefits of the change. The second column lists the incident types, and for each type the *incident value* is shown in the leftmost column. The incident values are shown on each change request screen, though they are the same for all.

The *incident frequency* values are entered in column 3, and we see that the need for this request comes from the numbers of service interruptions and callouts. With each frequency is associated a *frequency base* in column 4, given here as a code number to be interpreted according to the small table at the bottom right of the display.

Above this latter table is a button labelled "Edit." This, when clicked, reveals a small display of data editing buttons with instructions for their use.

7.4 Limitations of the Implementation

Firstly, it is unlikely that the HyperCard development system would be used in a practical implementation. Its flexibility and ease of graphic design have made it a good choice for the purposes of this thesis, but there is a penalty in terms of speed of operation. It copes well when asked to deal with a list of only eight change requests, but if that number were to increase perhaps into the hundreds the speed would be unacceptable.

A practical project to which the method was to be applied would in any case be expected already to have its own database system for change control. It would be much more reasonable to expect that the functionality would be added to the existing system, and that the (presumed) existence of a specialised

database package would provide the speed to handle the algorithms.

In entering the benefits associated with a change request, the implementation does not permit one *incident type* to be associated with more than one *frequency base*. If, for instance, a fault is incurring the same type of cost both per installation per year and per new installation, only one of these can be entered. This restriction allowed the screen display for change requests to be simplified.

The implementation does not allow for the *change delay* parameter. This would be required in a full implementation.

7.5 Chapter Summary

The model has been implemented on a Macintosh computer, using the HyperCard development system. Alternative plans are created as separate stacks (in the terminology of HyperCard; databases would be an approximately equivalent term). Exploration of plans is a matter of navigating around and between these plans, and navigation controls activated by the mouse are provided and have been explained.

The implementation has four types of display, of which two are used for data entry and two for the presentation of results. The data entry displays are for change requests and for the model's general parameters; there is a main summary screen with a graph of profit against release time, and for each release time examined there is a scenario display showing the detailed composition and outcome of a release timed at that point.

Chapter 8

Evaluation of the Model

8.1 Introduction

This chapter describes the use of the model with sample data. Although the model has been tested on real projects, there are two reasons why actual data from those projects will not be presented here:

- Full descriptions would contain many more change requests than are presented here, and would in general require many more pages of explanation in order not to distort their details. The added length would not enhance the description of the model's operation.
- Use of actual data would be commercially sensitive, and would restrict the final availability of the thesis as a whole.

The decision is therefore to base the chapter heavily on one particular project on which the model was tested, but to insert specimen figures that will illustrate

the points made. The numbers used in the examples do not therefore reflect actual or typical assessments of costs and benefits. The real project does have in common with the one presented here:

- Importance to the core business of the company. If this system fails, a lot of customers get very upset.
- Serving a large customer base.
- Significant distribution costs. When the installation of a software upgrade involves the replacement of memory boards at each installation, it gets expensive.

None of these attributes limits the generality of the model, but between them they do permit a wider range of behaviours to be explored.

Sections 8.2 to 8.2.3 set out the baseline against which the subsequent examples are to be compared. Section 8.2 describes the system generally, including the data that will be entered as the general parameters. Section 8.2.2 then presents the change requests that are assumed to be present.

Section 8.2.3 then presents the results of running the model on the sample data. The displays are shown, and a table gives a comparison of the model's conclusions for the range of possible release timings.

Each of Sections 8.3 to 8.7 then considers a variation of the baseline situation, and examines how the baseline plan would be affected.

Section 8.8 presents a summary of the results, comparing the assessments of the different variations.

8.2 The Baseline Plan

8.2.1 The Situation

Our hypothetical system is owned by a (presumably also hypothetical) telecommunications company and forms part of the telecomms network. It is a concentrator, which is the term for a small switch serving perhaps a few hundred telephone customers. The switch handles all local calls within its customer base, but is parented on a larger switch which deals with calls to and from wider areas. The parent switch also handles requests for most specialised services. The concentrator software probably consists of around 100,000 lines of code — about a quarter of a mile by Foster's metric. Within the concentrator, the software is held in read-only memory on a replaceable circuit board, so installing any upgrade means a site visit and a board replacement.

There are currently 3,500 installations in service, and that number is projected to rise to 4,700 over the next two years. After that, no further units are expected to be installed. None will be taken out of service during the next five years, which is as far ahead as our planning is expected to look.

The software is maintained by one person, who also has another project of similar size to look after. The ongoing effort available for this task is therefore 0.5 of a person. The cost of employment for a full-time person is £60,000 per year.

There are only eight change requests outstanding on the software, which is a small number for such a system but convenient for the purpose of discussion. In

partial compensation for that, each request will require quite substantial effort, ranging from 0.5 to 3 person-months. In addition, implementation of each change is associated with a certain amount of direct expenditure in re-issue of operational documentation etc: this ranges from £100 to £5,000.

The build process for such a system may take no more than a day, but the severe consequences of failures will dictate an extended test period before handover. Here, we assume that the build and test will take two months and require total expenditure (including staff time) of £100,000.

After the handover, a further period is required for acceptance testing and a pilot field trial, and for preparing for the full distribution. We assume this process takes a further two months and costs a total of £150,000.

Distribution then begins and it too will take time, dictated partly by the availability of staff to visit the sites and partly by the supply of freshly programmed memory boards. We will assume that in total, it will be possible to upgrade 200 sites each month at a cost of £100 each.

These values are entered on the general parameters screen, as shown in Figure 8.1.

8.2.2 The Change Requests

These are the requests:

1. **Data Change Speedup:** Improves the interface through which engineers enter configuration changes to the database. Will save about 6 minutes per installation per month. Cost estimate to fix: £1,200 + 1 person-month.

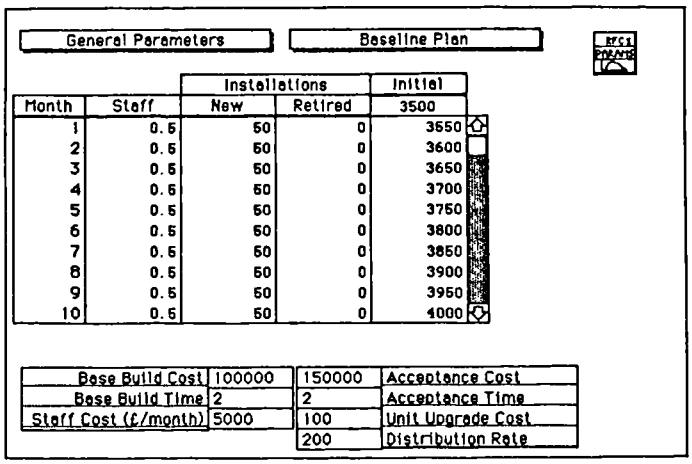

Figure 8.1: General Parameters Entered in the Model

2. **English Commands:** Some of the software was reused from a module written by a foreign software house, and engineers see some messages in that language. It's always obvious from the context what is meant, so in fact this one comes under the category of cosmetic change and no *measurable* benefit can be identified. Cost estimate to fix: £1,500 + 0.5 person-month.

3. **Customer Lockout:** Occasionally an individual customer's line will become locked up. Each time it happens, the customer complains and an engineer must be sent to reset the line. On average, each installation is suffering one of these failures every 20 months. Cost estimate to fix: £1,050 + 0.5 person-months.

4. **Dropped Calls:** Calls in progress are occasionally dropped because of this software error. It's not frequent: in a group of 25 installations, there will be about one such incident per month. Individual customers are

affected so rarely that no complaints have been recorded. Cost estimate to fix: £100 + 1 person-month.

5. **Transmission Loss:** This one subjects a small proportion of calls to sudden transmission loss: it happens on average once a month per installation. On 10% of these occasions, an engineer has to be sent out to investigate the problem. Cost estimate to fix: £5,000 + 2 person-months.

6. **Database Auto Load:** New installations have their initial configurations keyed in and checked by hand, which takes 6 hours. This enhancement would automate much of the process, saving 5 hours on each new installation. Cost estimate to fix: £3,000 + 1 person-month.

7. **One-Step Install:** Every new installation must be revisited after a few days to check calibration of power supply etc. This enhancement allows the checks to be made remotely, saving the expense of the extra visit. Cost estimate: £1,500 + 1 person-month.

8. **Bell Tinkle:** The system performs regular automatic tests on customers' lines, but unfortunately this sometimes causes the phenomenon of "bell tinkle." This modification redesigns the test procedure to eliminate the problem. At the moment, each installation generates one customer complaint roughly every 5 months. Cost estimate to fix: £1,500 + 3 person-months.

The data for each request is entered into its own screen. The example used in Chapter 7 (Figure 7.4) in fact shows the data for request number 5: Transmis-

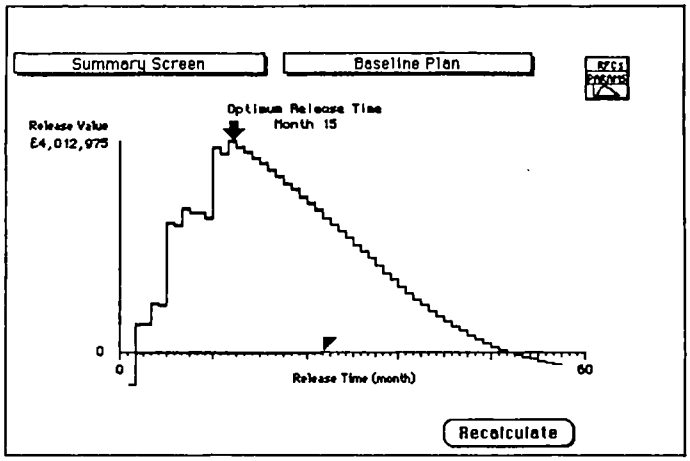

Figure 8.2: Summary screen showing profit against release date

sion Loss.

8.2.3 Baseline Plan Results

With the above data entered, the model is ready to run and Figure 8.2 shows the summary screen after calculations are complete. The peak profit of £4,012,975 occurs for a release date at the end of month 15, although there is another peak almost as high two months earlier. For the details, it is then necessary to examine the scenario graphs. Figure 8.3 shows the scenario for our best release after month 15, with on the right the list of requests for change. The lines of asterisks separate those that should be included (above) from those that should not (below).

All the changes except one ("English Commands") have positive priority, so there would have been seven changes included in the release if resources had allowed. As expected, the four that are included are the ones with the highest calculated priorities.

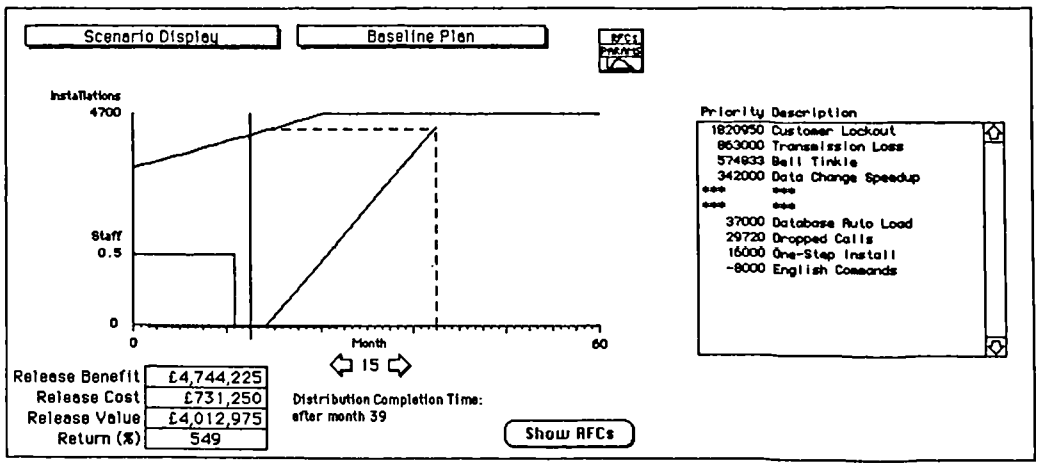

Figure 8.3: Scenario for release after month 15

If we wish to examine the slightly lower peak of profit for a release after month 13, we can look at that scenario (Figure 8.4). The profit figure is now £3,884,138 — a little over 3% below the 15-month figure. Only three changes can now be accommodated given the available resource, but the loss of benefit from "Data Change Speedup" is partially compensated by the reduced effort in the release, and by the increased benefit from the other changes through their earlier introduction into the field. The increased benefits generally from the earlier release have raised all the priority figures, except for "English Commands" where no benefit at all had been identified.

These two scenarios also illustrate a sensitivity to the criterion used for deciding which is to be the best release. Our decision to go for the highest profit, as we have seen, makes month 15 the best. If, however, the desire had been to maximise the return on investment we would have preferred to release after month 13, which in fact yields the highest return at 551%.

By examining each of the scenarios, a full picture may be built up of the

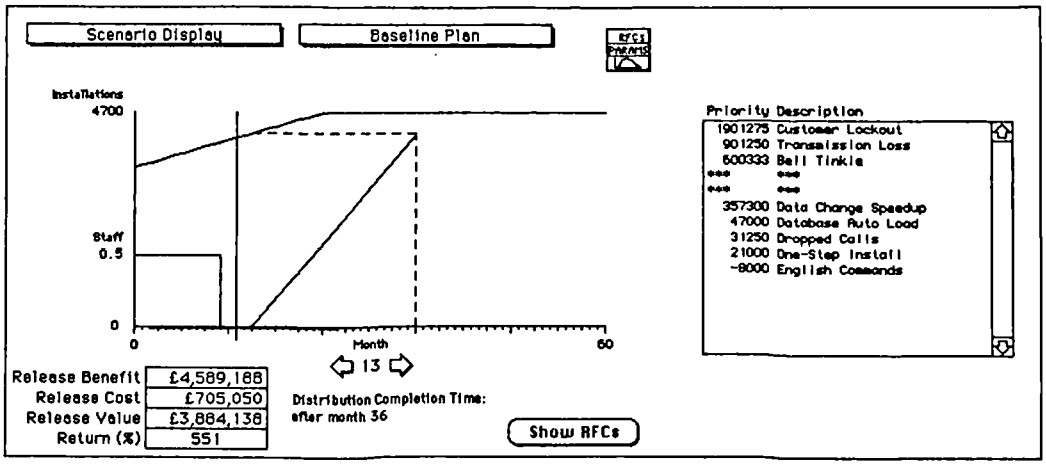

Figure 8.4: Scenario for release after month 13

implications of all the different possible release dates. On the model the natural way to do this is to use the arrow buttons to move back and forth, but here we will extract the principal items and show them as Table 8.1.

Referring to the table, we see that change 3 ("Customer Lockout") is always chosen for inclusion — a reassuring agreement with intuition. Change 1 ("Data Change Speedup") drops in and out of favour as the release date changes, because it gives way to number 5 ("Transmission Loss") and then to number 8 ("Bell Tinkle") when there is enough effort available to make these possible but not then enough still to include it.

Change 6 ("Database Auto Load") is of value only for new installations, of which there are no more after month 24. For releases after month 11 or 12, enough new installations remain to be done that it gets a high enough priority for inclusion; after that it remains out of the rankings until month 19, when the value (and hence priority) is much reduced but the resource is available to take in lower priority changes. After month 22 there are so few new installations yet

Release after month	Profit £	Return %	Changes included							
			1 DCS	2 EC	3 CL	4 DC	5 TL	6 DAL	7 OSI	8 BT
3	521,450	82			•					
4	502,025	79			•					
5	901,238	141	•		•					
6	873,688	137	•		•					
7	2,462,275	368			•		•			
8	2,402,800	359			•		•			
9	2,731,262	405	•		•		•			
10	2,662,262	395	•		•		•			
11	2,639,750	376	•		•		•	•		
12	2,564,025	365	•		•		•	•		
13	3,884,138	551			•		•			•
14	3,778,088	536			•		•			•
15	4,012,975	549	•		•		•			•
16	3,895,800	533	•		•		•			•
17	3,805,252	517	•		•	•	•			•
18	3,684,472	500	•		•	•	•			•
19	3,586,770	469	•		•	•	•	•		•
20	3,458,245	452	•		•	•	•	•		•
21	3,329,348	432	•		•	•	•	•	•	•
22	3,197,078	418	•		•	•	•	•		•
23	3,080,375	397	•		•	•	•			•
24	2,951,360	380	•		•	•	•			•
25	2,822,345	364	•		•	•	•			•
26	2,693,330	347	•		•	•	•			•

Table 8.1: Implications of various release dates (baseline plan)

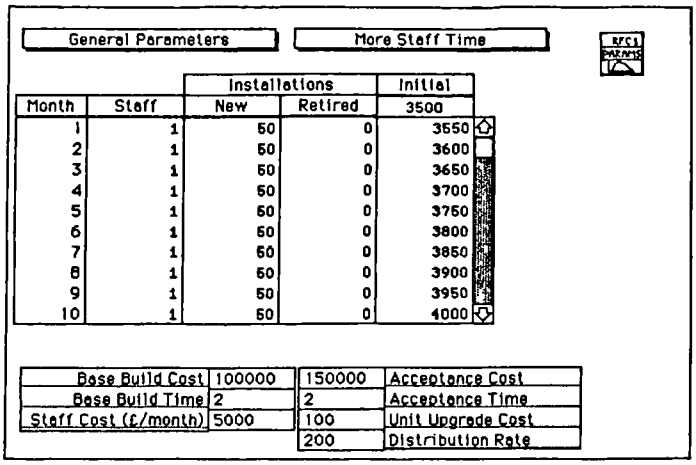

Figure 8.5: Increasing the available staff time

to be done that the benefit does not outweigh the cost, and so it drops again out of the list. Change 7 ("One-Step Install") undergoes a similar effect but from a lower base priority; it just makes it into the list after month 21, but after that it descends once more into oblivion.

8.3 More Staff Time

Clearly, the availability of the maintainer's time is putting severe limitations on the ability to deliver a timely release. Is it worth the extra salary cost of making our maintainer full time?

To examine the effect of taking this action, it is necessary only to alter the "Resource" column in the general parameters screen to show a constant value of 1 (see Figure 8.5).

Rerunning the model then leads to the new summary screen shown in Figure 8.6. The peak profit is now £4,689,262.5, which is an improvement of

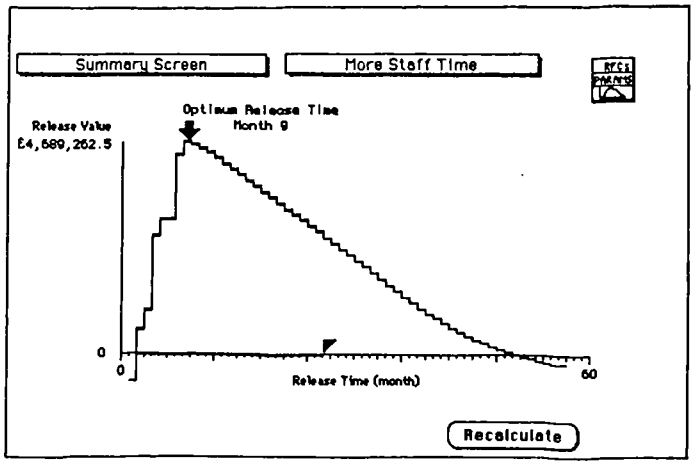

Figure 8.6: Summary with increased staff time

£676,287.5 over the baseline plan — nearly 17%. Moreover, the best release has been brought forward by six months, occurring now after month 9.

To see which changes are now included in the best release, we refer to the scenario graph for month 9, which is given in Figure 8.7. In what might be described as something of an anticlimax, we see that the same four are included as were in the baseline plan, so the extra profit has all come from the increased benefit of moving the release date forward.

The picture also reveals that not all the available resource has been used, as shown by the dashed vertical line within the resource histogram. In fact the spare resource shown there is half a person-month, so the release could have included another change of that effort or less. However, the only candidate is "English Commands" and its negative priority has precluded it.

It may be observed that the appropriate response in fact would be to bring the release forward by another two weeks, since the period during which the resource is spare is not used for any other purpose. That would be done in

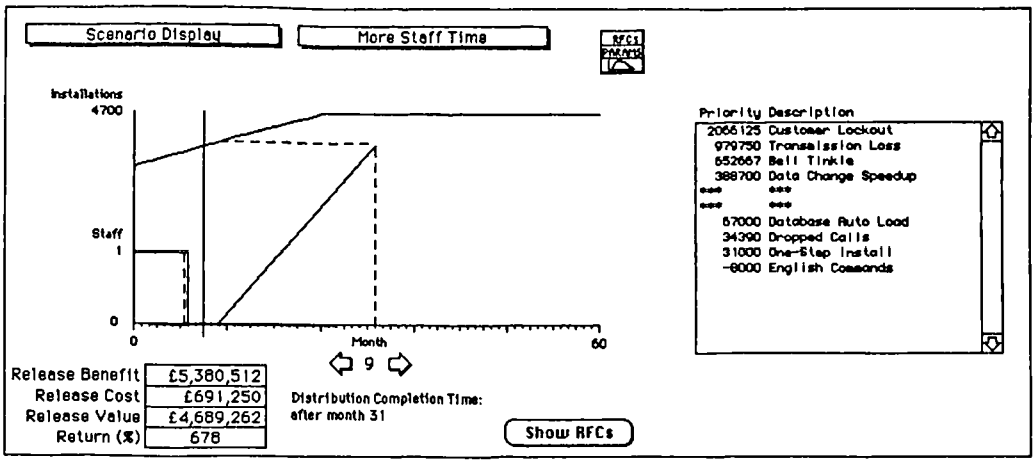

Figure 8.7: More staff time: Release after month 9

practice, but the model does not suggest it because one month is its minimum time resolution.

The return on investment for this release is given in Figure 8.7 as 678%, so by this measure also it is an improvement on the original.

8.4 Review of Testing

As an alternative to the resource increase, we might wish to explore the effect of spending money to improve the test process. The two months required for the build and system test, if it could be reduced, would certainly bring in more benefit through earlier release.

We will suppose that a specialised team can be hired for the system test. Their efforts will reduce the build and test time from the present 2 months to one, but the cost of their service will be £125,000. Should they be used?

In order to investigate this one, we return to the general parameters screen

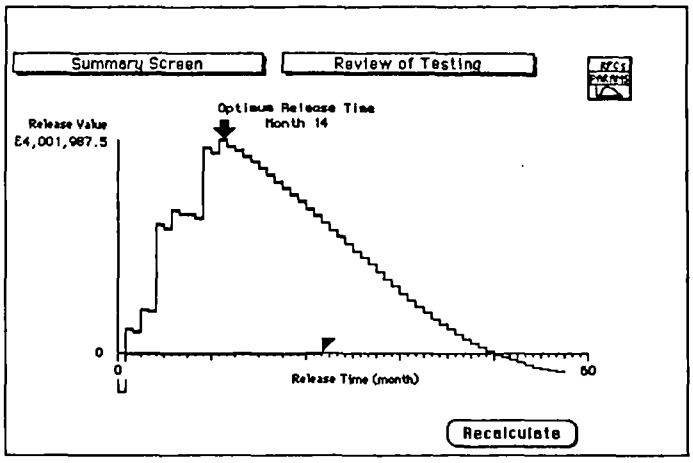

Figure 8.8: Effect of system test team

and return it to its baseline state as in Figure 8.1. Then, we add the £125,000 to the release preparation cost and reduce the release preparation time to 1.

Rerunning the model now gives Figure 8.8 as the summary screen. The peak profit is now a month earlier, as expected, but the actual value of the profit is £4,001,987.5 — £10,987.5 less than the baseline figure. Checking the scenario screen (Figure 8.9) shows that the same four changes are included.

The conclusion must be that the use of the test team is not justified. Despite the possibility of the earlier release, it is better to stick with the baseline plan — unless, of course, the cost of the test team can be negotiated down to an acceptable figure.

The decision in this case has been very marginal: it hinges on a difference of only 0.27% in the *release value*. If we had not made the simplifying assumption of constant value of money but instead assumed a future discount rate of $3\frac{1}{2}\%$ per annum or more, the test team would have been shown to be justified.

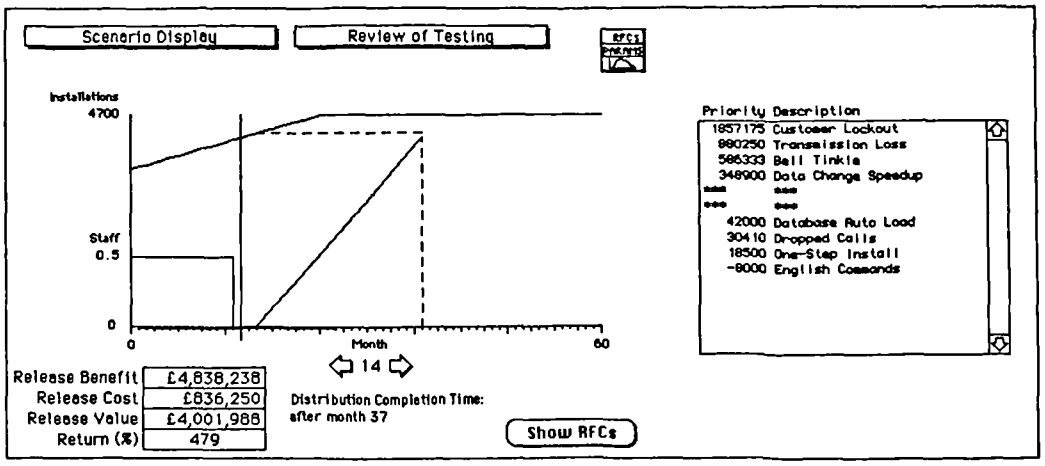

Figure 8.9: Test team: Release after month 14

8.5 System Replacement Plan

Under the assumptions of the baseline plan, no installations of the system are taken out of service during the five-year planning horizon. We now consider what would be the effect of an advance in technology, such that a replacement system renders ours obsolete within the five-year period. We will not be concerned with the economics of the new system: our need is simply to adjust the maintenance plan for the old one, if that should prove necessary.

We will assume that the new system will begin to enter service in two years time, so the installations of our system will be removed progressively from then. Starting in month 24, systems will be replaced at the rate of 125 units per month; from month 36 this figure will increase to 200. At these rates, all installations have been replaced by the end of month 50.

Is it still worth issuing another release of the software, or should we call it quits now?

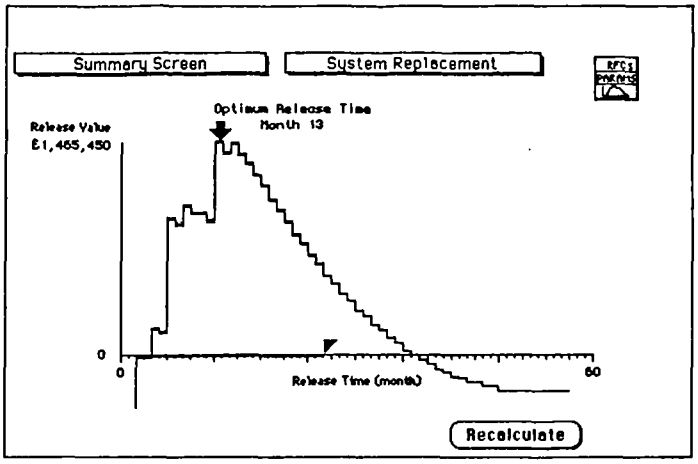

Figure 8.10: Effect of system replacement plan

Working again from the baseline figures, we enter the new data into the model by setting the new values into the *retired installations* field of the general parameters screen. Again we rerun the model, and we examine the new summary screen (Figure 8.10).

The effect of the new technology is clearly seen in the new peak profit figure, which is now down to £1,465,450. The best release is after month 13, two months earlier than the baseline plan. The scenario display (Figure 8.11) shows that the new return on investment is 242%, so although a new release is justified in theory by the still-positive profit it has become more likely that another project may show a better return on the expenditure.

Figure 8.11 also shows that the four changes that have always been included so far are now reduced to three. Change 1 ("Data Change Speedup") still has a relatively high priority, but on balance it has proved better to get the release out early than to wait while that particular change is implemented.

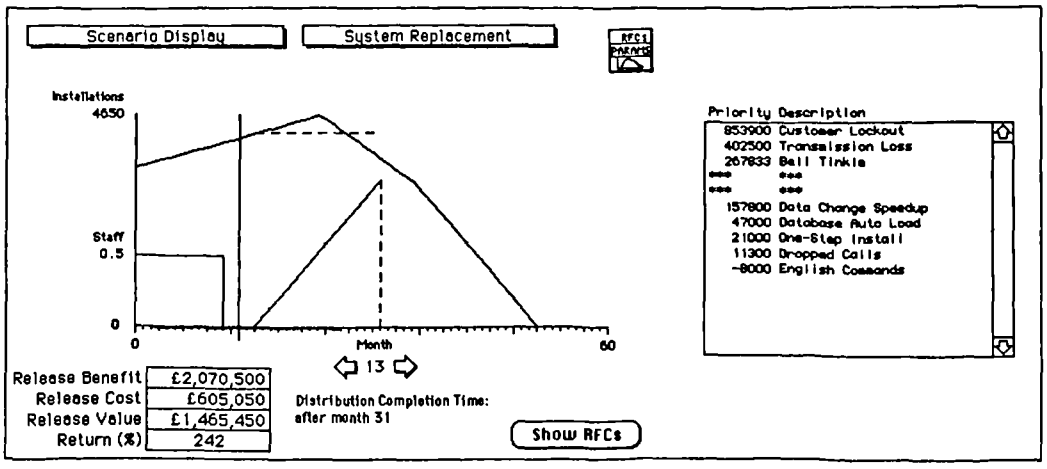

Figure 8.11: New technology: Release after month 13

8.6 New Legislation

So far, none of the situations we have examined has involved a mandatory change to the software. We will now rectify that.

New EC legislation is being enacted. Three years from now, it will become illegal for any company to ask its workforce to use software that does not communicate with them in their own native language.

Suddenly, the English Commands change takes on a whole new importance. Will this affect the baseline plan, and if so, how?

Returning to the baseline data, we set the "Force Selection" checkbox in the RFC screen for English Commands, and rerun the model. The new summary screen appears as Figure 8.12.

At first sight, the effect is not great. The best release is delayed by a month, reflecting the need to allow time for the design of the change, but it includes the same four changes from the baseline release plus the now mandatory English

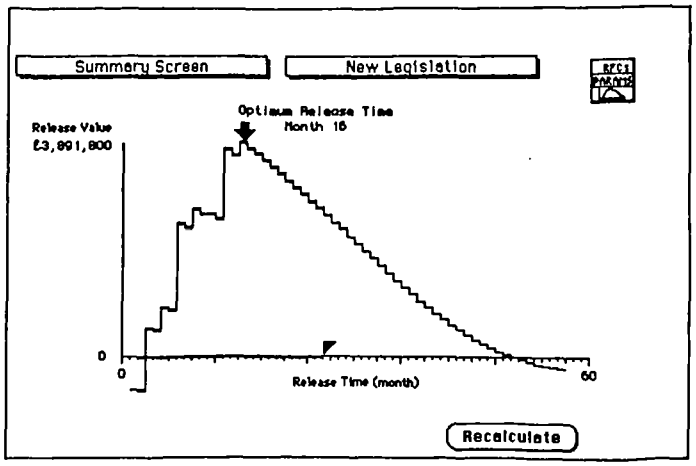

Figure 8.12: The effect of legislation

Commands. The scenario graph is shown in Figure 8.13. It is very similar to Figure 8.3 but English Commands has moved up to the top of the list.

Unfortunately though, the best release does not comply fully with the legislation because the new release will not have been installed at all sites within the required three years. Figure 8.13 shows this scenario, which misses the deadline by four months.

We therefore test earlier scenarios until we find one that gets the distribution finished in time, using the slide-show procedure described on page 98. That turns out to be a release after month 13 (Figure 8.14). Any release before that will also meet the legislative deadline, so we are then prepared to accept the best release at or before month 13. It turns out that there is a better one: a release after month 10, which turns in a profit of £2,658,262 (Figure 8.15). This is the one that should be accepted.

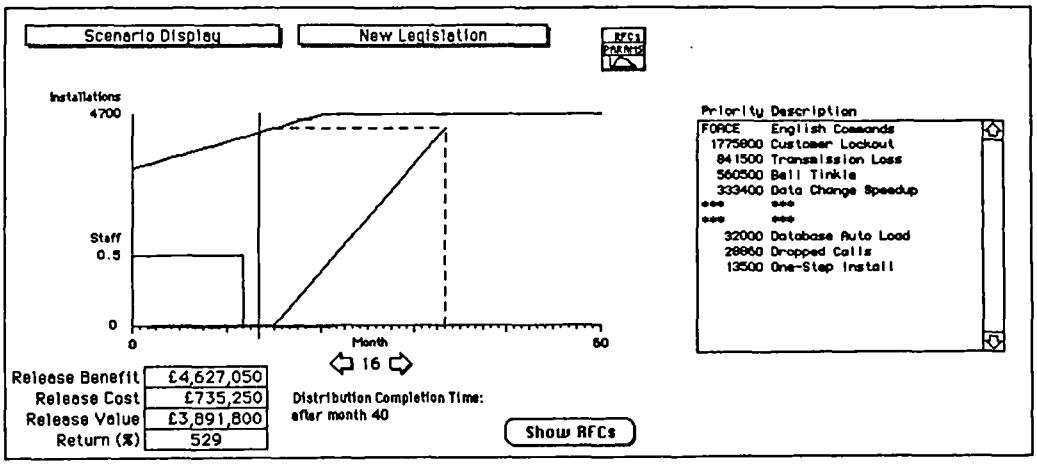

Figure 8.13: After legislation: The first estimate

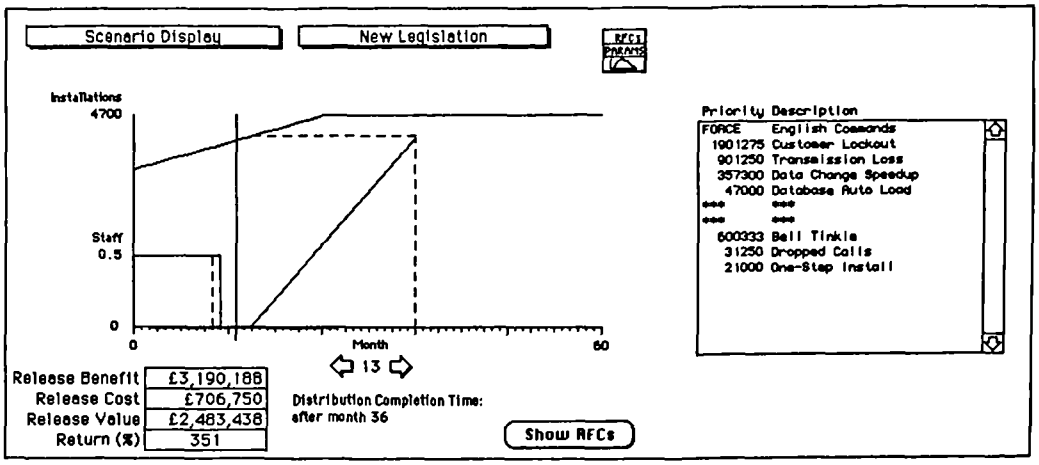

Figure 8.14: After legislation: Just in time

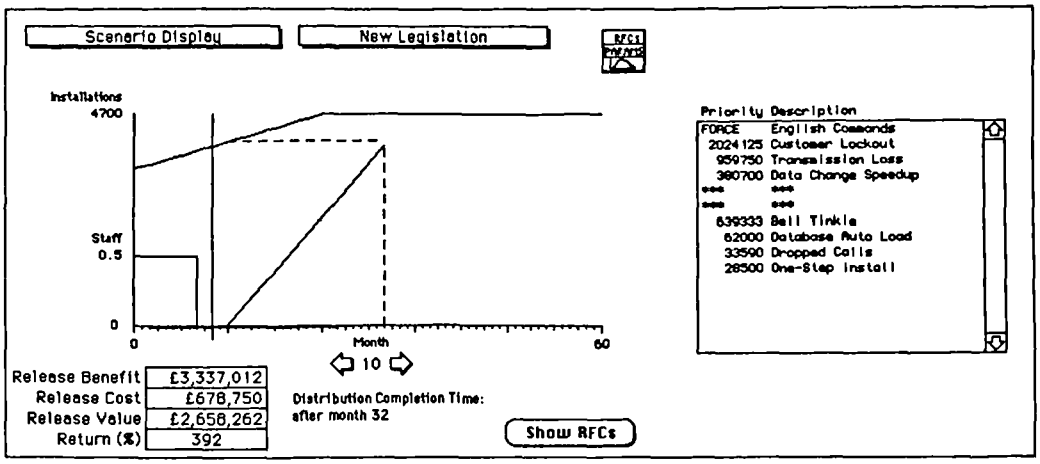

Figure 8.15: After legislation: The best solution

8.7 Distribution Review

For a typical release situation, more time is spent upgrading the installations in the field than in preparing the changes. This is a consequence of our initial assumption of an embedded system, but it would be expected in practice that distribution arrangements would be carefully reviewed to try and improve the system.

For this example, we will assume that such a review has been carried out. It would indeed be possible to speed things up, but only at a price.

The new scheme would mean assigning extra staff to the distribution task, and purchasing extra spare memory boards. As a result, the distribution rate can be doubled from the baseline figure of 200 installations per month to 400. However, there will be an initial outlay of £50,000 for equipment and training, and each upgrade will cost an additional £50.

These figures all affect the general parameters screen, leaving it as shown in

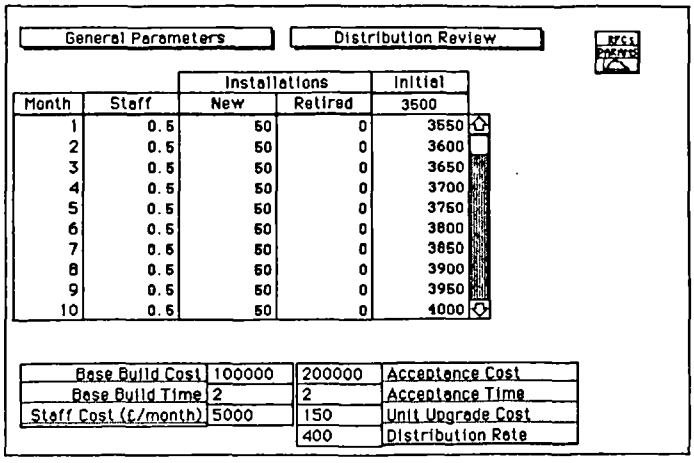

Figure 8.16: Distribution Review: General parameters

Figure 8.16. Rerunning the model with these figures then gives the summary screen as in Figure 8.17, from which we see that the new peak has a profit figure of £4,132,650. The scenario display for that peak, and the list of changes, are shown in Figure 8.18.

Figure 8.18 clearly shows the effect of the new distribution rate: the release is fully installed in the field at the end of month 28, as against month 38 in the baseline. The changes, and the release date itself, remain the same as in the baseline so no consequent planning changes are needed there. However, the profit figure of £4,132,650 is better by £119,675. Expensive though it is, the new distribution system is worthwhile.

8.8 Summary of Results

We have seen in the six scenarios how changes in the overall situation can affect the plans for the release of a new version of the software product, and the profit

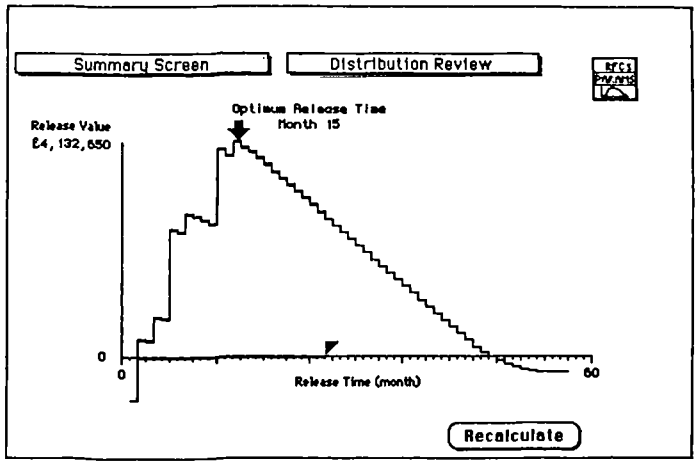

Figure 8.17: Distribution Review: Summary screen

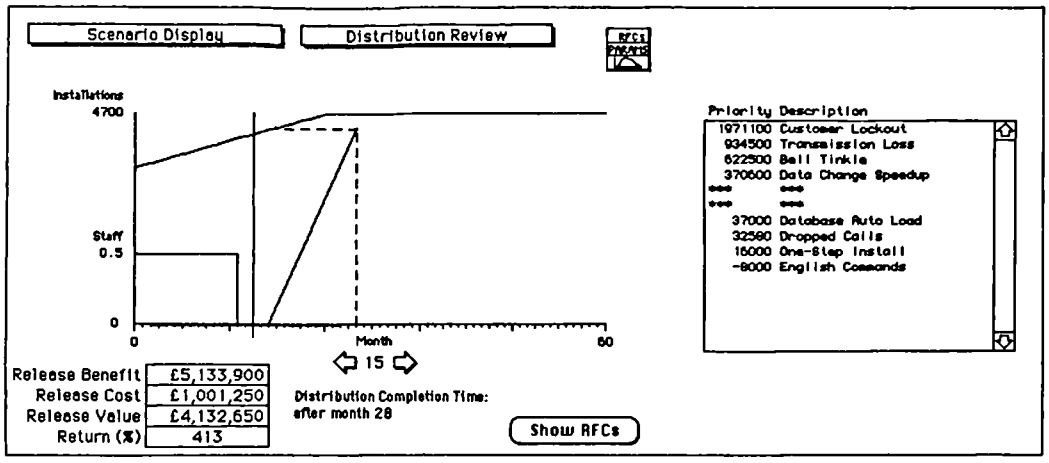

Figure 8.18: Distribution Review: Best scenario at month 15

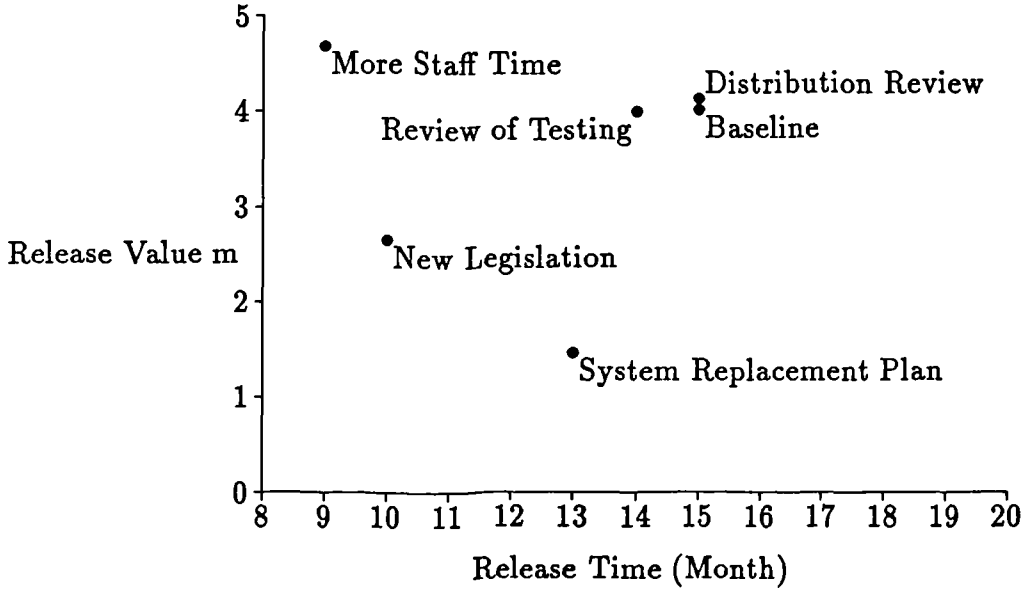

Figure 8.19: Summary of Results.

to be made from that release.

Figure 8.19 shows the six outcomes as a scatter plot of *release value* against *optimum release time.* Comparing each point with the baseline, we can see that the externally imposed events New Legislation and System Replacement Plan had the most dramatic effect on the *release value*, with the system replacement plan cutting profits by almost two thirds.

The three situations that could be said to be under the organisation's control, however, were more beneficial. The distribution review generated a small extra profit, while the review of testing brought the *release time* forward. In this case the profit was slightly reduced, though the effect is not apparent on the scale of the graph. We recall, too, that if the future value of money had been discounted in the calculations, then any annual discount rate of $3\frac{1}{2}\%$ or more would have justified the situation.

The greatest effect of all, however, is in the allocation of more staff time to the project. It brought the *release time* forward by a full six months, and yielded a useful increase in profit into the bargain. Discounting of the future value of money would further have increased the value of this situation.

8.9 Chapter Summary

This chapter has shown the use of the model based on hypothetical sample data, which however is based on data from a real project to which the model was applied.

The use of the model to provide a plan for a single situation was first demonstrated. It showed how the priority calculation would distinguish (by negative or positive priority) between those changes that could with advantage be included, but it further showed that positive priority is not in itself enough to gain inclusion in a release. The maintenance effort means that a change must also carry enough benefit to outweigh the release delay while it is designed, and many will not meet this condition.

The further use of the model to explore variations on the baseline situation was then demonstrated. The provision of extra resource for change design provided the greatest advantage, which accords well at least with this author's own experience.

Chapter 9

Conclusions

9.1 Summary of Thesis

The maintenance of computer software is a process which in an average organisation will consume more than half of the total software budget. Demand (in the form of change requests) normally outstrips supply (in the form of software changes in new releases), and prioritisation is an important component of the process. It is normal practice for organisations to allocate fixed budgets to maintenance activities, within which the target is to implement as many changes as possible from those with the highest priorities.

Existing models of the maintenance process concentrate on actions and entities. The thesis has presented a larger model, where the actions and entities are recognised at the lower levels but higher levels are increasingly concerned with costs, benefits and investment returns.

The thesis is concerned with the planning of future releases of software, and

the presented model is used in the derivation of calculations to yield financial implications of different plans. These calculations have been implemented as a computer program.

The program in turn is given a set of sample data to work on, and used to establish a baseline plan for a new release. Variations in the situation are applied, and the resulting effects on the plans are noted.

9.2 Results

Assessing the maintenance activity on an investment basis is significantly more difficult than for many other decisions, largely because many individual activities must be taken into account. Each of these has its own cost/benefit attributes, and the calculation of overall plans is repetitive and tedious: in fact, impractical without machine assistance.

The model has been applied primarily to one project within BT, involving an embedded system installed in large numbers within the network. The original figures have not been used in this thesis for reasons of commercial confidentiality, but sample figures have been substituted that allow similar conclusions to be drawn.

Although the model lacks some features that a practical implementation would require, those features would be feasible in a full implementation.

9.3 Statement of Success

Three criteria for success were stated at the start of this thesis.

> 1. *Describe the maintenance process with the aid of a quantified model that provides financial analysis of the consequences of proposed maintenance actions.*

We have developed a 7-level model of the maintenance process, which is described in Chapter 4. Aspects of the model which are relevant to release planning are formalised in Chapter 5. The model predicts the outcome of maintenance situations, in financial terms and with timescales. The model goes beyond the immediate needs of the thesis, and is a suitable basis for further developments. The comparison of alternate investment values will need to be tailored for the organisation that uses the model, and in Section 5.12 we have indicated how this would be done.

> 2. *Show that the model aids decision making both within a maintenance project and in the comparison of investment values between projects.*

The model aids decision-making, as introduced in Chapter 6 and as demonstrated by example in Chapter 8. The examples in Chapter 8 illustrate the comparison of alternative situations by reference to their investment returns. Investment returns are absolute financial values, which may be compared amongst different projects including non-software ones. The presentation of information

from the model may be improved after further study, and in Section 9.4.2 we indicate the possibility of future work in this area.

3. *Show that the model can be implemented on a computer and that the implementation has the potential of practical commercial use.*

The model has been implemented in a computer program, and while the current implementation is more suitable as a demonstration than for direct commercial use, we have indicated in Section 7.4 the changes that would be required.

9.4 Further Work

9.4.1 Research

Further research work based on this thesis can be as follows:

1. The model works on the basis of a release at some future time, yet it makes no allowance for the arrival of further change requests during that time. Data on past arrivals could be collected and extrapolated on any particular project, but there remains the possibility that more general studies could reveal the existence of typical patterns. Knowledge of these could give the model a measure of independence from that form of data collection.

2. The nature of the model makes it inevitable that its predictions should be subject to sudden change in response to gradual changes in values of

input parameters. This aspect of its behaviour would benefit from further study, which would be aimed at predicting the bounds of the magnitudes of the discontinuities.

3. The sample situations tested in Chapter 8 include examples of predicting the effects of process improvements. This kind of use of the model can be investigated further. For example, if historical information about past change requests is available, it would be possible to drive the model by assuming future requests would follow a similar pattern. This would make it possible to provide longer-term predictions about the effects of process change.

4. Knowledge of any typical statistical distributions in the inputs to the model may improve its predictive powers and its general application. If distributions can be identified, then further theoretical work becomes a possibility.

9.4.2 Exploitation

Further work to aid the exploitation of the results of this thesis can be as follows:

1. The model allows for a variety of calculations of return on investment, and a calculation would need to be specified for the most favoured value. This would depend on the normal practices of the organisation using the model.

2. The model has been evaluated by exposing it to members of project teams and gaining their approval in principle. It has not been used over a period of time, and an extended trial would be a necessary part of further evaluation.

3. The model has so far been exposed to projects which have dealt with embedded systems. It needs to be tested in other environments such as with data processing systems where (for example) the distribution costs and times may prove relatively insignificant.

4. The model has been exposed only to planning for small software changes, in accordance with the usual software maintenance definition. There appears to be no reason why it could not deal with the planning of large changes such as major enhancements. An analysis of sample business cases would test this suggestion, and could increase the scope for exploitation. It would also suggest that the definition of maintenance should be re-examined, since the distinction between small and large changes is at present so large a component of it.

5. It is likely that the presentation of information from the model could be improved. For example, the presentation in Figure 8.19 is not generated automatically. It could be.

9.5 Conclusions

In conclusion, the following main points have been expanded in this thesis:

1. Software maintenance, unlike development, is commonly regarded as a regrettable but necessary expense. It has been demonstrated that maintenance can be treated as an investment activity, in which benefits are weighed against costs and financial returns calculated.

2. This information can be used to guide maintenance planning decisions.

3. The necessary calculations can be embodied in a computer program, and it is practicable to use such a program to explore the consequences of alternative scenarios.

Bibliography

[BO75] C. L. Brantley and Y. R. Osajima. Continuing development of centrally developed and maintained software systems. *IEEE Computer Society Proceedings*, (45):285–288, 1975.

[Boe76] Barry W. Boehm. Software engineering. *IEEE Trans. Comput.*, 25(12):1226–1241, December 1976.

[Boe86] B. W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5), May 1988.

[Bro75] F. P Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass., 1975.

[Can72] R. G. Canning. That maintenance iceberg. *EDP Analyzer*, 10(10):1–14, October 1972.

[CB86] James S. Collofello and Stephen Bortman. An analysis of the technical information necessary to perform effective software maintenance. In *Proc. 5th Annual Phoenix Conference on Computers and Communications*, pages 420–424, March 1986.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CC92] Larry D. Cousin and James S. Collofello. A task-based approach to improving the software maintenance process. In *Proc. 8th IEEE Conf. Software Maintenance*, pages 118–126, November 1992.

[Cha86] Ned Chapin. Supervisory attitudes toward software maintenance. In *Proc. AFIPS National Computer Conference*, pages 61–68, 1986.

[Cle91] E. Clemons. Evaluation of strategic investments in information technology. *CACM*, 34(1):22–36, January 1991.

[Cra62] W. J. Craig, editor. *The Complete Works of William Shakespeare.* Oxford University Press, 1962.

[Dat73] Program maintenance: User's view. *Data Processing*, 7:1–4, 1973.

[DBSB91] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: A knowledge-based software information system. *CACM*, 34(5):34–49, May 1991.

[Dre92] Daniel W. Drew. Tailoring the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to a software sustaining engineering organisation. In *Proc. 8th IEEE Conf. Software Maintenance*, pages 137–144, November 1992.

[DSA71] A. E. Ditri, J. C. Shaw, and W. Atkins. *Managing the EDP Function.* McGraw–Hill, New York, 1971.

[Els76] James L. Elshoff. An analysis of some commercial PL/1 programs. *IEEE Trans. SE*, 2(2):113–120, June 1976.

[EM82] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *CACM*, 25(8):512–521, August 1982.

[FH79] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. Technical report, GUIDE 48, Philadelphia, PA, 1979.

[Fle83] James C. Fletcher. Report of the study on eliminating the threat posed by nuclear ballistic missiles. Technical report, SDI Organisation, 1983.

[FM87] John R. Foster and Malcolm Munro. A documentation method based on cross-referencing. In *Proc. 3rd IEEE Conf. Software Maintenance, Austin, Texas*, September 1987.

[Fos89] J. R. Foster. Priority control in software maintenance. In *Proc. 7th Int. Conf. Software Engineering for Telecommunication Switching Systems, Bournemouth, UK*, pages 163–167, July 1989.

[Fos91] J. R. Foster. Program lifetime: A vital statistic for maintenance. In *Proc. 7th IEEE Conf. Software Maintenance*, October 1991.

[Fro85] David Frost. Software maintenance and modifiability. In *Proc. Conf. Computers and Communications*, pages 489–494, 1985.

[Gre84] Lee L. Gremillion. Determinants of program repair maintenance requirements. *CACM*, 27(8):826–832, August 1984.

[Gui83] Tor Guimaraes. Managing application program maintenance expenditures. *CACM*, 26(10):739–746, October 1983.

[HB92] D. S. Hinley and Keith H. Bennett. Developing a model to manage the software maintenance process. In *Proc. 8th IEEE Conf. Software Maintenance*, pages 174–182, November 1992.

[HQ92] Del-Raj Harjani and Jean-Pierre Queille. A process model for the maintenance of large space system software. In *Proc. 8th IEEE Conf. Software Maintenance*, pages 127–136, November 1992.

[Hum89] Watts S. Humphrey. *Managing the Software Process.* Addison-Wesley, 1989.

[IEE93] IEEE Computer Society. *IEEE Standard for Software Maintenance (IEEE Std 1219-1993)*, 1993.

[J. 73] J. Hoskyns and Co. Implications of using modular programming. Hoskyns System Research, London, 1973. Guide no. 1.

[Jon77] T. Capers Jones. Program quality and programmer productivity. Technical Report TR 02.764, IBM, 1977.

[Jon81] Robert R. Jones. Software continuation engineering system concept. In *Proc. Nat. Conf. on Software Technology and Management, Alexandria, VA*, October 1981.

[Kel92] Ted W. Keller. The importance of process improvement in software maintenance (keynote address). In *Proc. 8th IEEE Conf. Software Maintenance*, 1992.

[Ken90] Robert C. Kendall. More management perspectives on programs, programming and productivity. *Software Maintenance News*, 8(5):22, May 1990.

[Kha75] Zafar Khan. How to tackle the systems maintenance dilemma. *Canadian Datasystems*, pages 30–32, March 1975.

[Liu76] C. C. Liu. A look at software maintenance. *Datamation*, 22(11):51–55, November 1976.

[LMR91] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. In *Proc. 7th IEEE Conf. Software Maintenance*, pages 171–178, October 1991.

[LS80] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley, Reading, MA, 1980.

[LST78] Bennet P. Lientz, E. Burton Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *CACM*, 21(6):466–471, June 1978.

[McC81] Carma L. McClure. *Managing Software Development and Maintenance.* Van Nostrand Reinhold Co., New York, NY, 1981.

[McD92] Jim McDonald. Ensuring the benefits of process improvement. In *Proc. Sixth European Software Maintenance Workshop, Durham, UK*, September 1992.

[Mil75] H. D. Mills. How to write correct programs and know it. *ACM SIGPLAN Notices*, 10:363–370, June 1975.

[Mor88] Robert Moreton. Analysis and results from a maintenance survey. In *Proc. Second Software Maintenance Workshop, Durham, UK*, September 1988.

[MW79] J. H. Morrissey and L. S.-Y. Wu. Software engineering: An economic perspective. In *Proc. 4th Int. Conf. Software Eng., Munich, West Germany, 1979*, pages 412–422, September 1979.

[Nau76] Report on the NATO Software Engineering Conference, Garmisch, Germany, 1968. In Peter Naur and Brian Randall, editors, *Software Engineering Concepts and Techniques.* Petrocelli/Charter, 1976.

[NP90] John T. Nosek and Prashant Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3):157–174, September 1990.

[Rig69] R. Riggs. Computer system maintenance. *Datamation*, 15:227–235, November 1969.

[RU89] H. Dieter Rombach and Bradford T. Ulery. Establishing a measurement based maintenance improvement program: Lessons learned in the SEL. In *Proc. 5th IEEE Conf. Software Maintenance*, pages 50–57, October 1989.

[Sha77] W. K. Sharpley. Software maintenance planning for embedded computer systems. In *Proc. COMPSAC '77*, pages 520–526, November 1977.

[Sta84] Thomas A. Standish. An essay on software reuse. *IEEE Trans. SE*, 10(5):494–497, September 1984.

[Swa79] E. Burton Swanson. On the user-requisite variety of computer application software. *IEEE Proc. Reliability*, 28:221–226, August 1979.

[Til87] Mike Tilley. Debrief report on the National Computer-Aided Software Engineering Conference October 1987. Technical memo, British Telecom, October 1987.

[TT92] Tetsuo Tamai and Yohsuke Torimitsu. Software lifetime and its evolution process over generations. In *Proc. 8th IEEE Conf. Software Maintenance*, November 1992.

[vZ93] H. J. van Zuylen, editor. *The REDO Compendium.* John Wiley & Sons, 1993.

[WDK93] R. E. Whiting, J. Davies, and M. Knul. Investment appraisal for IT systems. *British Telecom Technical Journal*, 11(2):193–211, April 1993.

[WH91] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. In *Proc. 7th IEEE Conf. Software Maintenance*, pages 162–170, October 1991.

[Zel78] M. V. Zelkowitz. Perspectives on software engineering. *ACM Computing Surveys*, 10(2):197–216, June 1978.

Index of Model Terms