

**SOFTWARE EVOLUTION AND THE STAGED MODEL OF THE  
SOFTWARE LIFECYCLE**

K. H. Bennett

Research Institute for Software Evolution

University of Durham

UK

DH1 3LE

[keith.bennett@durham.ac.uk](mailto:keith.bennett@durham.ac.uk)

V. T. Rajlich

Department of Computer Science

Wayne State University

Detroit, MI 48202

[vtr@cs.wayne.edu](mailto:vtr@cs.wayne.edu)

N. Wilde

Department of Computer Science

University of West Florida

Pensacola, FL 32514

[nwilde@uwf.edu](mailto:nwilde@uwf.edu)

## Table of Contents

|   |    |
|---|----|
| ABSTRACT .....                                  | 4  |
| 1. Introduction .....                           | 5  |
| 1.1 Background .....                            | 5  |
| 1.2 Early work .....                            | 8  |
| 1.3 Program comprehension .....                 | 11 |
| 1.4 Standards .....                             | 11 |
| 1.5 Iterative software development .....        | 13 |
| 1.6 The laws of software evolution .....        | 13 |
| 1.7 Stage distinctions .....                    | 15 |
| 1.8 The business context .....                  | 16 |
| 1.9 Review .....                                | 19 |
| 1.10 The stages of the software lifecycle ..... | 22 |
| 2. Initial Development .....                    | 23 |
| 2.1 Introduction .....                          | 23 |
| 2.2 Software team expertise .....               | 24 |
| 2.3 System architecture .....                   | 25 |
| 2.4 What makes architecture evolvable? .....    | 26 |
| 3. Evolution - The Key Stage .....              | 27 |
| 3.1 Introduction .....                          | 28 |
| 3.2 Software releases .....                     | 29 |
| 3.3 Evolutionary software development .....     | 30 |
| 4. Servicing .....                              | 33 |
| 4.1 Software decay .....                        | 33 |
| 4.2 Loss of knowledge and cultural change ..... | 34 |

|  |    |
|--|----|
| 4.3 Wrapping, patching, cloning, and other “kludges” ..... | 36 |
| 4.4 Re-engineering .....                                   | 38 |
| 5. Phase-Out and Close-Down .....                          | 39 |
| 6. Case Studies .....                                      | 41 |
| 6.1 The Microsoft Corporation .....                        | 42 |
| 6.2 The VME operating system .....                         | 43 |
| 6.3 The Y2K experience .....                               | 44 |
| 6.4 A major billing system .....                           | 46 |
| 6.5 A small security company .....                         | 47 |
| 6.6 A long-lived defense system .....                      | 48 |
| 6.7 A printed circuits program .....                       | 49 |
| 6.8 Project PET .....                                      | 50 |
| 6.9 The FASTGEN geometric modeling toolkit .....           | 52 |
| 6.10 A financial management application .....              | 53 |
| 7. Software Change and Comprehension .....                 | 55 |
| 7.1 The miniprocess of change .....                        | 55 |
| 7.2 Change request and planning .....                      | 56 |
| 7.3 Change implementation .....                            | 58 |
| 7.4 Program comprehension .....                            | 62 |
| 8. Sustaining Software Value .....                         | 67 |
| 8.1 Staving off the servicing stage .....                  | 67 |
| 8.2 Strategies during development .....                    | 69 |
| 8.3 Strategies during evolution .....                      | 74 |
| 8.4 Strategies during servicing .....                      | 77 |
| 9. Future Directions: Ultra rapid software evolution ..... | 78 |

|                        |    |
|------------------------|----|
| 10. Conclusions .....  | 84 |
| Acknowledgements ..... | 85 |
| References .....       | 86 |

## **ABSTRACT**

Software maintenance is concerned with modifying software once it has been delivered and has entered user service. Many studies have shown that maintenance is the dominant lifecycle activity for most practical systems; thus maintenance is of enormous industrial and commercial importance. Over the past 25 years or so, a conventional view of software development and maintenance has been accepted in which software is produced, delivered to the user, and then enters a maintenance stage. A review of this approach and the state of the art in research and practice is given at the start of the chapter.

In most lifecycle models, software maintenance is lumped together as one phase at the end. In the experience of the authors, based on how maintenance is really undertaken (rather than how it might or should be done), software maintenance actually consists of a number of stages, some of which can start during initial development. This provides a very different perspective on the lifecycle. In the chapter, we introduce a new model of the lifecycle that partitions the conventional maintenance phase in a much more useful, relevant and constructive way. It is termed the *staged model*.

There are five stages through which the software and the development team progress. A project starts with *initial development stage*, and we then identify an explicit *evolution stage*. Next is a *servicing stage*, comprising simple tactical activities. Later still, the software moves to a *phase-out stage* in which no more work is done on the software other than to collect revenue from its use. Finally the software has a *close-down stage*. The key point is that software evolution is quite different and separate from servicing, from phase-out, and from close-down, and this distinction is crucial in clarifying both the technical and business consequences.

We show how the new model can provide a coherent analytic approach to preserving software value. Finally, promising research areas are summarized.

## **1. Introduction**

### **1.1 Background**

What is software maintenance? Is it different from software evolution? Why isn't software designed to be easier to maintain? What should we do with legacy software? How do we make money out of maintenance? Many of our conventional ideas are based on analyses carried out in the 1970s and it is time to rethink these for the modern software industry.

The origins of the term maintenance for software are not clear, but it has been used consistently over the past 25 years to refer to post-initial delivery work. This view is reflected in the IEEE definition of software maintenance [1] essentially as a post delivery activity:

*The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.*

Implicit in this definition is the concept of software life cycle, which is defined as [1]:

*The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. Note: These phases may overlap or be performed iteratively.*

A characteristic of established engineering disciplines is that they embody a structured, methodical approach to developing and maintaining artifacts [2, chapter 15, chapter 27]. Software lifecycle models are abstract descriptions of the structured methodical development and modification process typically showing the main stages in producing and maintaining executable software. The idea began in the 1960s with the waterfall model [3].

A lifecycle model, implicit or explicit, is the primary abstraction that software professionals use for managing and controlling a software project, to meet budget, timescale and quality objectives, with understood risks, and using appropriate resources. The model describes the production of deliverables such as specifications and user documentation, as well as the executable code. The model must be consistent with any legal or contractual constraints within a project's procurement strategy. Thus it is not surprising that lifecycle models have been given primary attention within software engineering community.

A good overview of software lifecycle models is given in [2] and a very useful model is the spiral model of Boehm [4], which envisages software production as a continual iterative development process. But crucially, this model does not address the *loss of knowledge*, which in the authors' experience accompanies the support of long-lived software systems and which vitally constrains the tasks which can be performed. Our aim was to create a lifecycle model that would be useful for the planning, budgeting and delivery of evolving systems, and that would take into account this loss of knowledge. Our new model is called the *staged model*.

The aim of this chapter is to describe the new staged model [5]. We provide a broad overview of the state of the art in software maintenance and evolution. The emphasis is mainly on process and methods, (rather than technology), since this is where the main developments have occurred, and

is of most relevance to this chapter. There is much useful material available on software maintenance management, including very practical guides [6]. We start from the foundations established within the international standards community. We then briefly revisit previous research work, as an understanding of these results is essential. Program comprehension is identified as a key component; interestingly, very few textbooks on software engineering and even on software maintenance mention the term, so our review of the state of the art addresses the field to include this perspective. The new model and our view of research areas are influenced by program comprehension more than other aspects. The staged model is presented, and evidence drawn from case studies. Practical implications are then described, and finally, research directions are presented.

## **1.2 Early work**

In a very influential study, Lientz and Swanson [7][8] undertook a questionnaire survey in the 1970s, in which they analyzed then-current maintenance practices. Maintenance changes to software were categorized into:

- perfective (changes to the functionality)
- adaptive (changes to the environment)
- corrective (the correction of errors)
- preventive (improvements to avoid future problems).

This categorization has been reproduced in many software engineering text books and papers (e.g. Sommerville [9], McDermid [2], Pressman [10], Renaissance [11] etc.), and the study has been repeated in different application domains, in other countries and over a period of 20 years (see for example the Ph.D. thesis of Foster [12], who analyses some 30 studies of this type). However, the basic analysis has remained essentially unchanged, and it is far from clear what benefits this view of maintenance actually brings.

Implicit in the Lientz and Swanson model are two concepts:

- That software undergoes initial development, it is delivered to its users, and then enters a maintenance phase
- The maintenance phase is uniform over time in terms of the activities undertaken, the process and tools used, and the business consequences.

These concepts have also suggested a uniform set of research problems to improve maintenance, see for example [2, chapter 20]. One common message emerging from all these surveys is the very substantial proportion of lifecycle costs that are consumed by software maintenance, compared to software development. The figures range from 50% to 90% of the complete lifecycle cost [7]. The proportion for any specific systems clearly depends on the application domain and the successful deployment of the software (some long lived software is now over 40 years old!).

The balance of lifecycle costs is subject to commercial pressures. It may be possible to discount the purchase price of a new software system, if charges can be recovered later through higher maintenance fees. The vendor may be the only organization capable (i.e. having the knowledge and expertise) of maintaining the system. Depending on the contractual arrangement between the producer and the consumer, and on the expectations for the maintenance phase, the incentive during development to produce maintainable software may vary considerably. Software maintenance is thus not entirely a technical problem.

In the 1990s, the practice of outsourcing software maintenance became widespread. A customer company subcontracts all the support and maintenance of a purchased system to a specialist subcontractor. This has raised a set of new commercial issues, for example the potential risk of subcontracting out key company systems; and the difficulties in the future of recalling maintenance back in house or to another subcontractor if the first does not perform acceptably.

It is important to recall that it is not simply the software application that evolves. Long lived software may well outlast the environment within which it was produced. In the military domain, software has sometimes lasted longer than the hardware on which it was cross compiled (presenting major problems if the software has to be modified). Software tools are often advocated for software maintenance, but these may also evolve (and

disappear from the market) at a faster rate than the software application under maintenance.

### **1.3 Program comprehension**

Program comprehension is that activity by which software engineers come to an understanding of the behavior of a software system using the source code as the primary reference. Studies suggest that program comprehension is the major activity of maintenance, absorbing around 50% of the costs [2, chapter 20][13]. Program comprehension requires understanding of the user domain that the software serves as well as software engineering and programming knowledge of the program itself. Further details are given in Section 7.

The authors believe that comprehension plays a major role in the software lifecycle. During the early stages, the development team builds group understanding, and the system architects have a strategic understanding of the construction and operation of the system at all levels. At later stages, this knowledge is lost as developers disperse and the complexity of the software increases, making it more difficult to understand. Knowledge appears impossible to replace, once lost, and this forms the basis for our new model.

### **1.4 Standards**

Software maintenance has been included within more general software engineering standardization initiatives. For example, the IEEE has published a comprehensive set of standards [14], of which Std. 1219 on maintenance forms a coherent part. The IEEE standard defines seven steps in software maintenance change:

- Problem modification/identification, classification and prioritization
- Analysis and understanding (including ripple effects)
- Design
- Implementation
- Regression/system testing
- Acceptance testing
- Delivery.

Underpinning the standard is a straightforward iterative perspective of software maintenance; a change request is reviewed and its cost estimated; it is implemented; and then validation is carried out.

The International Standards Organization ISO has also published a software maintenance standard [15]. This is in the context of Std. ISO/IEC 12207 which addresses how an agreement should be drawn up between a software acquirer and supplier (in which maintenance is included). The standard places considerable emphasis on planning.

## 1.5 Iterative software development

The iterative nature of software lifecycle was noted already in 70's by several authors. Niklaus Wirth [16] proposed Stepwise Refinement where functionality is introduced into the program in successive iterations. Basili and Turner [17] described another process where the functionality is added to the program in successive iterative steps.

Large software projects of that time already followed iterative scenarios. A notable project was the development of the IBM OS operating system. The experience of that project was described in [18] and in [19]. These authors noted that software life cycle is inherently iterative, that a substantial proportion of the functionality is added iteratively and that the initial development is simply the initialization stage of this process. See also [20].

## 1.6 The laws of software evolution

The evolution of a software system conforms to laws, which are derived from empirical observations of several large systems [21][22][23]:

1. *Continuing change.* A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful.

2. *Increasing complexity.* As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.

The laws are only too apparent to anyone who has maintained an old heavily changed system. Lehman also categorized software into three types, as follows [24]:

**S type software:** this has a rigid specification (S means *static*) which does not change and which is well understood by the user. The specification defines the complete set of circumstances to which it applies. Examples are offered by many mathematical computations. It is therefore reasonable to prove that the implementation meets the specification.

**P type software:** this has a theoretical solution, but the implementation of the solution is impractical or impossible. The classic example is offered by a program to play chess, where the rules are completely defined, but do not offer a practical solution to an implementation (P means *practical*). Thus we must develop an approximate solution which is practical to implement.

**E type software:** this characterizes almost all software in everyday use, and reflects the real world situation that change is inevitable (E means *embedded*, in the sense of embedded in the real world). The solution involves a model of the abstract processes involved, which includes the software. Thus the system is an integral part of the world it models, so

change occurs because both the world changes and the software is part of that world.

For E type systems, Lehman holds the view that (as in any feedback system), the feed-forward properties such as development technologies, environments and methods are relatively unimportant and global properties of the maintenance process are insensitive to large variations in these factors. In contrast, he argues that the *feedback* components largely determine the behavior. So issues like testing, program understanding, inspection and error reports are crucial to a well understood stable process. This work may provide a theoretical perspective of why program comprehension is so important. He also amplified the concept of E type software in his principle of uncertainty: *the outcome of software system operation in the real world is inherently uncertain with the precise area of uncertainty also unknowable*. This work to establish a firm scientific underpinning for software evolution continues in the FEAST project ([25][26] [27][28]).

### **1.7 Stage distinctions**

Sneed [29] and Lehner [30][31] are among the few authors to have observed that the maintenance phase is not uniform. Sneed classified systems into three categories: *throw-away systems* that typically have a lifetime of less than two years and are neither evolved nor maintained. Then there are *static systems* that are implemented in a well-defined area and after being

developed, their rate of change is less than 10% in a year, or are relatively static after development. Finally there are *evolutionary systems* that undergo substantial change after the initial development and last many years. The stages of these systems are initial development, evolution (called “further development” by Sneed), “maintenance” (i.e. servicing), and “demise” (i.e. phase-out).

Lehner [30] used this model and investigated the lifecycle of 13 business systems from Upper Austria to confirm the existence of the stages. He found that some systems belong to the category of static systems, where there is a very short evolution after the initial development and then the maintenance work very dramatically decreases. Other systems consume substantial effort over many years. Lehner confirmed a clear distinction between evolution (called “growth” in his paper) and servicing (called “saturation”) where the maintenance effort is substantially lower [31]. He thus refuted earlier opinions that the evolution and growth of software can continue indefinitely and confirmed Sneed’s earlier observation about the distinctions between the several stages, through observation of long-term data from several systems.

## **1.8 The business context**

Some of the problems of the traditional lifecycle model stem from recent trends in the business context of software development. Different categories

of software application are subjected to radically different kinds of business pressures.

Most software engineering techniques available today for software specification, design, and verification have been presented as conventional supply-side methods, driven by technological advance. Such methods may work well for systems with rigid boundaries of concern, such as embedded systems, which may be characterized as risk-averse. In such domains, users have become familiar with long periods between requests for new features and their release in new versions (the so-called “applications backlog”).

However such techniques break down for applications where system boundaries are not fixed and are subject to constant urgent change. These applications are typically found in emergent organizations - “organizations in a state of continual process change, never arriving, always in transition” [32]. Examples include *e-businesses* as well as more traditional companies that continually need to reinvent themselves to gain competitive advantage [33]. A stockbroker, for example, may have a need to introduce a new service overnight; the service may only exist for another 24 hours before it is replaced by an updated version. In such organizations we have a demand-led approach to the provision of software services, addressing delivery mechanisms and processes which, when embedded in emergent organizations, give a software solution in emergent terms - one with continual change. The solution never ends and neither does the provision of

software. The user demand is for change in “internet time” and the result is sometimes termed *engineering for emergent solutions*.

Yet a third category is provided by so-called "legacy systems", which have been defined [34] as “systems which are essential to our organization but we don’t know what to do with them”. They pose the epitome of the maintenance challenge, because for the great majority, remedial action has never taken place, so whatever structure originally existed has long since disappeared. Legacy systems have been extensively addressed in the literature (see e.g. [35][36]). The main conclusion is that there is no magic silver bullet; the future of each system needs to be analyzed, planned and implemented based on both technical and business drivers, and taking into account existing and future staff expertise.

Finally, as software systems become larger and more complex, organizations find that it does not make sense to develop in-house all the software they use. Commercial-Off-The-Shelf (COTS) components are becoming a larger part of most software projects. Selecting and managing such software represents a considerable challenge, since the user becomes dependent on the supplier, whose business and products may evolve in unexpected ways. Technical concerns about software's capabilities, performance, and reliability may become legal and contractual issues, and thus even more difficult to resolve.

It appears that many organizations are now or soon will be running, at the same time, a mixture of embedded or legacy software, with added COTS components, and interfaced to new e-business applications. The processes and techniques used in each category clash, yet managers need somehow to make the whole work together to provide the services that clients demand.

## **1.9 Review**

We began by drawing on currently available evidence. Many ideas are now sufficiently mature that process standards have been defined or are emerging, at least for large scale, embedded risk averse systems. Empirical studies have shown that program comprehension is a crucial part of software maintenance, yet it is an activity that is difficult to automate and relies on human expertise. The increasing business stress on time to market in emergent organizations is increasing the diversity of software types that must be managed, with different processes being appropriate for each type.

We conclude:

- Human expertise during the maintenance phase represents a crucial dimension that cannot be ignored. At the moment, some of the hardest software engineering evolution tasks (such as global ripple analysis) need senior engineers fully to comprehend the system and its business role.
- We need to explore maintenance to reflect how it is actually done, rather than prescriptively how we would like it to be done.

The major contribution of this chapter is to propose that maintenance is not a single uniform phase, the final stage of the conventional lifecycle, but is comprised of several distinct stages and is in turn distinct from evolution. The stages are not only technically distinct, but also require a different business perspective. Our work is motivated by the fact that the Lientz and Swanson approach does not accord with modern industrial practice, based on analysis of a number of case studies. Knowing the fraction of effort spent on various activities during the full lifecycle does not help a manager to plan those activities or make technical decisions about multi-sourced component-based software, or address the expertise requirements for a team. The conventional analysis has not, over many years, justified the production of more maintainable software despite the benefits that should logically accrue. Especially, the conventional model does not sensibly apply to the many modern projects, which are heavily based on COTS technology. Finally it is only a technical model and does not include business issues.

We also had other concerns that the conventional Lientz and Swanson model did not elucidate. For example, there are few guidelines to help an organization assess if a reverse engineering project would be commercially successful, despite the large amount of research and development in this field, and it was not clear why this was the case. The skills of staff involved in post-delivery work seem very important, but the knowledge needed both by humans and in codified form has not been clearly defined, despite a large number of projects which set out to recapture such knowledge.

Our motivation for defining a new perspective came from the very evident confusion in the area. A brief examination of a number of web sites and papers concerned with software engineering, software evolution and software maintenance also illustrated the confusion between terms such as maintenance and evolution (with a completely uniform maintenance phase), and the almost universal acceptance of the Lientz and Swanson analysis.

We found this situation inadequate for defining a clear research agenda that would be of benefit to industry. For these reason, we shall largely restrict our use of the term 'software maintenance' from now on in this chapter to historical discussions.

We have established a set of criteria for judging success of our new perspective:

- it should support modern industrial software development which stresses time to delivery and rapid change to meet new user requirements.
- it should help with the analysis of COTS type software.
- it should be constructive and predictive - we can use it to help industry to recognize and plan for stage changes
- it should clarify the research agenda - each stage has very different activities and requires very different techniques to achieve improvement

- it should be analytic - we can use it to explain and clarify observed phenomena, e.g. that reverse engineering from code under servicing is very hard
- it should be used to model business activity as well as technical activity.
- it should transcend as far as possible particular technologies and application domains (such as retail, defense, embedded etc.) while being applicable to modern software engineering approaches.
- it should also transcend detailed business models and support a variety of product types. On one hand we have the shrink-wrap model where time to market etc. are important considerations; at the other extreme we have customer-tailored software where the emphasis may be on other attributes like security, reliability, ease of evolution, etc.
- it should be supported by experimental results from the field.
- it should help to predict and plan, rather than simply be descriptive.

Our perspective is amplified below and is called the *staged model*.

### **1.10 The stages of the software lifecycle**

The basis of our perspective is that software undergoes several distinctive stages during its life. The following stages are:

- **Initial development** - the first functioning version of the system is developed.

- **Evolution** - if initial development is successful, the software enters the stage of evolution, where engineers extend its capabilities and functionality, possibly in major ways. Changes are made to meet new needs of its users, or because the requirements themselves had not been fully understood, and needed to be given precision through user experience and feedback. The managerial decision to be made during this stage is when and how software should be released to the users (alpha, beta, commercial releases, etc.)
- **Servicing** - the software is subjected to minor defect repairs and very simple changes in function (we note that this term is used by Microsoft in referring to *service packs* for minor software updates).
- **Phase-out** - no more servicing is being undertaken, and the software's owners seek to generate revenue for its use for as long as possible. Preparation for migration routes is made.
- **Closedown** - the software is withdrawn from the market, and any users directed to a replacement system if this exists.

The simplest variant of the staged software life cycle is shown in Figure 1.

<Figure 1 near here>

In the following sections, we describe each of these stages in more detail.

## 2. Initial Development

### 2.1 Introduction

The first stage is *initial development*, when the first version of the software is developed from scratch. This stage has been well described in the software engineering literature and there are very many methods, tools, and textbooks that address it in detail (for example, see [9][2][10][37]). The stage is also addressed by a series of standards by IEEE and ISO, or by domain specific or industry specific standards (for example in the aerospace sector). Such initial development very rarely now takes place starting from a ‘green field’ situation since there may be an inheritance of old legacy software, as well as external suppliers of new COTS components.

Over the past 30 years, since the recognition of software engineering as a discipline [38], a great deal of attention has been paid to the process of initial development of reliable software within budget and to predictable timescales. Software project managers welcomed the earliest process model, called the *waterfall model*, because it offered a means to make the initial development process more visible and auditable through identifiable deliverables. Since there is such an extensive literature dealing with initial development, we will cover only selected aspects of it.

## **2.2 Software team expertise**

From the point of view of the future stages, several important foundations are laid during initial development. The first foundation is that the expertise of the software engineering team and in particular of the system architects is established. Initial development is the stage during which the team learns

about the domain and the problem. No matter how much previous experience had been accumulated before the project started, new knowledge will be acquired during initial development. This experience is of indispensable value in that it will make future evolution of the software possible. So this aspect – the start of team learning – characterizes the first stage. Despite the many attempts to document and record such team learning, much of it is probably *tacit* – it is the sort of experience that is extremely difficult to record formally.

### **2.3 System architecture**

Another important result and deliverable from initial development is the architecture of the system, i.e. the components from which the system is built, their interactions, and their properties. The architecture will either facilitate or hinder the changes that will occur during evolution and it will either withstand those changes or break down under their impact. It is certainly possible to document architecture, and standard approaches to architectures (e.g. [39]) provide a framework.

In practice, one of the major problems for architectural integrity during initial development is “requirements creep”. If the requirements of the software system are not clear, or if they change as the software is developed, then a single clear view of the architecture is very difficult to sustain.

Numerous approaches to ameliorating this problem have been devised, such as rapid application development, prototyping, and various management

solutions, such as the Chief Programmer team, and (more recently) extreme programming [40]. The approach chosen can be strongly influenced by the form of legal contract between the vendor and customer which may induce either a short-term or a long-term view of the trade-off between meeting a customer's immediate needs and maintaining a clean software architecture.

#### **2.4 What makes architecture evolvable?**

Thus for software to be easily evolved, it has to have an appropriate architecture, and the team of engineers has to have the necessary expertise. For example in long-lived systems such as the ICL VME operating system, almost all sub-components have been replaced at some stage or another. Yet despite this, the overall system has retained much of its architectural integrity [41]. In our experience, evolution of architecture needs individuals of very high expertise, ability and leadership. There may be financial pressures to take technical shortcuts in order to deliver changes very quickly (ignoring the problem that these conflict with the architectural demands). Without the right level of human skill and understanding it may not be realized that changes are seriously degrading the software structure until it is too late.

There is no easy answer or “prescription” to making an architecture easily evolvable. Inevitably there is a trade off between gains now, and gains for the future, and the process is not infallible. A pragmatic analysis of software systems which have stood the test of time (e.g. VME, or UNIX) typically

shows the original design was undertaken by one, or a few, highly talented individuals.

Despite a number of attempts, it has proved very difficult to establish contractually what is meant by maintainable or evolvable software and to define processes that will produce software with these characteristics. At a basic level, it is possible to insist on the adoption of a house style to programming, to use IEEE or ISO standards in the management and technical implementation, to use modern tools, to document the software, and so on. Where change can be foreseen at design time, it may be possible to parameterize functionality.

These techniques may be necessary, but experience shows that they are not sufficient. The problem may be summarized easily: a successful software system will be subjected to changes over its lifetime that the original designers and architects cannot even *conceive* of. It is therefore not possible to plan for such change, and certainly not possible to create a design that will accommodate it. Thus some software will be able to evolve; but other systems will have an architecture that is at cross purposes with a required change. To force the change may introduce technical and business risks and create problems for the future.

### **3. Evolution - The Key Stage**

### 3.1 Introduction

The evolution stage is characterized as an iterative addition, modification, or deletion of nontrivial software functionality (program features).

This stage represents our first major difference from the traditional model.

The usual view is that software is developed and then passed to the maintenance team. However, in many of the case studies described later, we find that this is not the case. Instead, the software is released to customers, and assuming it is successful, it begins to stimulate enthusiastic users. (If it is not successful, then the project is cancelled!) It also begins to generate income and market share. The users provide feedback and requests for new features.

The project team is living in an environment of success, and this encourages the senior designers to stick around and support the system through a number of releases. In terms of team learning, it is usually the original design team that sees the new system through its buoyant early days. Of course, errors will be detected during this stage, but these are scheduled for correction in the next release. During the evolution stage, the continued availability of highly skilled staff makes it possible to sustain architectural integrity. Such personnel would seem to be essential. Unfortunately we note that making this form of expertise explicit (in a textbook, for example) has not been successful despite a number of projects concerned with "knowledge-based software engineering".

The increase in size, complexity and functionality of software is partly the result of the learning process in the software team. Cusumano and Selby reported that a feature set during each iteration may change by 30% or more, as a direct result of the learning process during the iteration [42].

Brooks also comments that there is a substantial "learning curve" in building a successful new system [18]. Size and complexity increases are also caused by customers' requests for additional functionality and market pressures add further to growth, since it may be necessary to match features of the competitor's product.

In some domains, such as the public sector, legislative change can force major evolutionary changes, often at short notice, that were never anticipated when the software was first produced. There is often a continuous stream of such changes.

### **3.2 Software releases**

There are usually several releases to customers during the software evolution stage. The time of each release is based on both technical and business considerations. Managers must take into account various conflicting criteria, which include time to market or time to delivery, stability of software, fault rate reports, etc. Moreover the release can consist of several steps, including alpha and beta releases. Hence the release, which

is the traditional boundary between software development and software maintenance, can be blurred and to a certain degree an arbitrary milestone.

For software with a large customer base, it is customary to produce a sequence of versions. These versions coexist among the users and are independently serviced, mostly to provide bug fixes. This servicing may take the form of patches or minor releases so that a specific copy of the software in the hands of a user may have both a version number and release number. The releases rarely implement a substantial new functionality; that is left to the next version. This variant of staged life cycle model for this situation is shown in Figure 2.

<Figure 2 near here>

### **3.3 Evolutionary software development**

The current trend in software engineering is to minimize the process of initial development, making it into only a preliminary development of a skeletal version or of a prototype of the application. Full development then consists of several iterations, each adding certain functionality or properties to the already existing software system. In this situation, software evolution largely replaces initial development, which then becomes nothing more than the first among several equal iterations.

The purpose of evolutionary development is to minimize requirements risks. As observed earlier, software requirements are very often incomplete

because of the difficulties in eliciting them. The users are responsible for providing a complete set of accurate requirements, but often provide less than that, because of the lack of knowledge or plain omissions. On top of that, the requirements change during development, because the situation in which the software operates changes. There is also a process of learning by both users and implementers and that again contributes to changing requirements. Because of this, a complete set of requirements is impossible or unlikely in many situations so one-step implementation of large software carries a substantial risk. Evolutionary software development that is divided into incremental steps lessens the risk because it allows the users to see and experience the incomplete software after each iteration.

One of the well-known and well-described processes of evolutionary software development is the Unified Software Development Process [43]. This process describes in detail how software is to be developed in incremental iterations. Each incremental iteration adds a new functionality or a new property (e.g. security, effectiveness) to the already existing software. This gradual increase in requirements lessens the risk involved, because each iteration provides a fresh feedback about the progress of the project. The Unified Software Development Process describes number of activities and specifies the documents to be produced during the iterations.

However Booch reports that a major criticism leveled at the Unified Software Development Process and similar approaches is that the resulting

processes are rigid, require extensive documentation and many steps, and consequently are too expensive in time for many modern businesses [44].

An emerging alternate approach for systems that require rapid evolution is the *agile method*, an example of which is Extreme Programming (XP) [40]. XP almost abolishes the initial development phase. Instead, programmers work closely with customers to develop a set of "stories" describing desired features of the new software. Then a series of releases is implemented, with typically only a few weeks between releases. The customer defines the next release by choosing the stories to implement. Programmers take the stories and define more fine-grained tasks, with one programmer taking responsibility for each. Test cases are defined before programming begins.

An interesting aspect of XP is that the responsible programmer signs up a partner for the task; all work is done in pairs with both working at the same workstation. Thus knowledge is shared between at least two programmers and some self-checking is built in without requiring organized walkthroughs or inspections. Pairs are broken up and reformed for different tasks so experience can be distributed. There is little documentation of code or design though considerable care is taken to maintain tests that can be rerun in the future.

Agile methods seem to discard all the Software Engineering experience of the past twenty years and place their reliance purely on the retention of expert team personnel for as long as the software needs to evolve. They thus

gain valuable time, but perhaps at considerable risk. It remains to be seen if this kind of methodology will be viable beyond the short term or if managers and stockholders will instead discover that their critical applications have suddenly made unplanned and costly transitions to servicing.

## **4. Servicing**

### **4.1 Software decay**

As previously mentioned, software enters the servicing stage as human expertise and/or architectural integrity are lost. Servicing has been alternatively called "saturation" [30][31], "aging software", "decayed software", "maintenance proper", and "legacy software". During this stage, it is difficult and expensive to make changes, and hence changes are usually limited to the minimum. At the same time, the software still may have a "mission critical" status, i.e. the user organization may rely on the software for services essential to its survival.

Code decay (or aging) was discussed in [45] and empirical evidence for it was summarized in [46]. The symptoms of code decay include:

- excessively complex (bloated) code, i.e. code that is more complex than it needs to be
- vestigial code that supports features no longer used or required
- frequent changes to the code

- history of faults in the code
- delocalized changes are frequent, i.e. changes that affect many parts of the code
- programmers use “kludges”, i.e. changes done in an inelegant or inefficient manner, for example clones or patches
- numerous dependencies in the code. As the number of dependencies increases, the secondary effects of change become more frequent and the possibility of introducing an error into software increases.

#### **4.2 Loss of knowledge and cultural change**

In order to understand a software system, programmers need many kinds of knowledge. The programmers must understand the domain of the application in detail. They must understand the objects of the domain, their properties and relationships. They must understand the business process that the program supports, as well as all activities and events of that process. They also must understand the algorithms and data structures that implement the objects, events, and processes. They must understand the architecture of the program and all its strengths and weaknesses, imperfections by which the program differs from an ideal. This knowledge may be partially recorded in program documentation, but usually it is of such a size and complexity that a complete recording is impractical. A great part of it usually is not recorded and has the form of individuals' experiences or groups' oral tradition.

This knowledge is constantly at risk. Changes in the code make knowledge obsolete. As the symptoms of decay proliferate, the code becomes more and more complicated and larger and deeper knowledge is necessary in order to understand it. At the same time, there is usually a turnover of programmers on the project.

Turnover may have different causes, including the natural turnover of the programmers for their personal reasons, or the needs of other projects that forces managers to reassign programmers to other work. Based on the success of the project, team members are promoted, moved to other projects, and generally disperse. The team expertise to support strategic changes and evolution to the software is thus lost; new staff members joining the team have a much more tactical perspective (e.g. at code level) of the software. Evolvability is lost and, accidentally or by design, the system slips into servicing.

However, management that is aware of the decline may recognize the eventual transition by planning for it. Typically, the current software is moved to the servicing stage, while the senior designers initiate a new project to release a radically new version (often with a new name, a new market approach etc.).

A special instance of the loss of knowledge is cultural change in software engineering [47]. Software engineering has almost half century of tradition and there are programs still in use that were created more than 40 years ago.

These programs were created in a context of completely different properties of hardware, languages and operating systems. Computers were slower and had much smaller memories, often requiring elaborate techniques to deal with these limitations. Program architectures in use were also different; modern architectures using techniques such as object orientation were rare at that time. The programmers who created these programs are very often unavailable. Current programmers who try to change these old programs face a double problem: not only do they have to recover the knowledge that is necessary for that specific program, but they also have to recover the knowledge of the culture within which it and similar programs were created. Without that cultural understanding they may be unable to make the simplest changes in the program.

#### **4.3 Wrapping, patching, cloning, and other “kludges”**

During servicing, it is difficult and expensive to make changes, and hence changes are usually limited to the minimum. The programmers must also use unusual techniques for changes, the so-called “kludges”. One such technique is *wrapping*. With wrapping, software is treated as a black box and changes are implemented as wrappers where the original functionality is changed into a new one by modifications of inputs and outputs from the old software. Obviously, only changes of a limited kind can be implemented in this way. Moreover each such change further degrades the architecture of the software and pushes it deeper into the servicing stage.

Another kind of change that frequently is employed during servicing is termed *cloning*. If programmers do not understand fully the program, instead of finding where a specific functionality is implemented in the program, they create another implementation. Thus a program may end up having several implementations of identical or nearly identical stacks or other data structures, several implementations of identical or almost identical algorithms, etc.

Sometimes programmers intentionally create clones out of fear of secondary effects of a change. As an example, let us assume that function  $f_{00}()$  requires a change, but  $f_{00}()$  may be called from other parts of the code so that change in  $f_{00}()$  may create secondary effects in those parts. Since knowledge of the program in the servicing stage is low, the programmers choose a “safe” technique: they copy-and-paste  $f_{00}()$ , creating an intentional clone  $f_{001}()$ . Then they update  $f_{001}()$  so that it satisfies the new requirements, while the old  $f_{00}()$  still remains in use by other parts of the program. Thus there are no secondary effects in the places where  $f_{00}()$  is called. While programmers solve their immediate problem in this way, they negatively impact the program architecture and make future changes harder.

The presence of a growing number of clones in code is a significant symptom of code decay during servicing. Several authors have proposed methods of detecting clones automatically using substring matching [48] or subtree matching [49]. Software managers could consider tracking the

growth of clones as a measure of code decay, and consider remedial action if the system seems to be decaying too rapidly [50,51].

Servicing patches are fragments of the code, very often in binary form, that are used to distribute bug fixes in a widely distributed software system.

#### **4.4 Re-engineering**

In the servicing stage, it is difficult to reverse the situation and return into the stage of evolution. That would require regaining the expertise necessary for evolution, recapturing the architecture, restructuring the software, or all of these. Both restructuring and regaining expertise are slow and expensive processes, with many risks involved, and new staff may have to be recruited with appropriate and relevant skills.

As analyzed by Olsem [52], the users of a legacy system build their work routines and expectations based on the services it provides and are thus very sensitive about any disruption of routine. Their tolerance of changes may be much smaller than the tolerance displayed by the users of brand new systems. Thus user rigidity also makes re-engineering a very risky proposition.

In order to minimize the risk and the disruption of user routine, Olsem advocates incremental re-engineering, where the system is re-engineered one part at a time. The new parts temporarily coexist with the old parts and

old parts are replaced one-by-one, without interruption of the service. A case study of such reengineering was published in [53].

This approach to re-engineering avoids disruption of the user's routines, but it also preserves the interfaces among the parts and hence the over-all architecture of the system. If the architecture is also obsolete, the process provides only partial relief. This impacts the business case for re-engineering, since the benefits returned compared to the investment required may be difficult to justify. In the worst case, we are spending resources for little or no benefit.

A further difficulty with re-engineering of widely-used software is the problem of distribution. Getting the new version out to all users can be expensive or impossible, so the burden of servicing the old version may persist. This problem will surely become even greater as software is increasingly introduced to more consumer goods such as mobile phones. Once object level code has been released in such devices it is all but impossible to go back to the evolutionary stage.

Based on our experience, complete re-engineering as a way of stepping back from servicing to evolution is very rare and expensive, so that entrance into the servicing stage is for all practical purposes irreversible.

## **5. Phase-Out and Close-Down**

At some stage the system is essentially frozen and no further changes are allowed. This stage, which we call phase-out, has also been called "decline" [31]. Help desk personnel may still be in place to assist users in running the system, but change requests are no longer honored. Users must work around any remaining defects instead of expecting them to be fixed. Finally, the system may be completely withdrawn from service and even this basic level of staffing is no longer provided.

The exact course of phase-out and closedown will depend on the specific system and the contractual obligations in place. Sometimes a system in phase-out is still generating revenue, but in other cases (such as most shrink-wrap software) the user has already paid for it. In this second case, the software producer may be much less motivated to provide support.

In a survey by Tamai and Torimitsu [54], an investigation was undertaken of the life span of software in Japan. The survey dealt with software from several application areas such as manufacturing, financial services, construction, mass media, etc. It found that for software larger than 1 million lines of code, the average life was 12.2 years with standard deviation of 4.2 years. The lifetime varied more widely for smaller software.

Tamai and Torimitsu's work also classified the causes of the closedowns in the following way. Hardware and/or system change caused the closedown in 18% of the cases. New technology was the reason in 23.7% of the cases. Need to satisfy new user requirements (that the old system was unable to

satisfy) was the cause in 32.8% of the cases. Finally deterioration of software maintainability was the culprit in 25.4% of the cases. We can speculate that at the end of the lifetime, the software was in phase-out stage and in most of the cases, there was an event (hardware change, new technology, new requirements) that pushed software into close-down. Only in 25.4% of the cases, did closedown occur naturally as a free management decision, without any precipitating event from the outside.

There are a number of issues related to software shutdown. Contracts should define the legal responsibilities in this phase. In some cases, such as outsourced software in which one company has contracted with another to develop the system, the relationships may be quite complex. Final ownership and retention of the system, its source code, and its documentation, should be clearly defined.

Frequently system data must be archived and access must be provided to it. Examples of such data are student transcripts, birth certificates, and other long-lived data. The issues of data archiving and long-term access must be solved before the system is shut down.

## **6. Case Studies**

Our new lifecycle model was derived from involvement with and observation of real industrial and commercial software development projects in a number of domains. We then abstracted from the particular

experiences and practices of these projects, in order to draw our new perspective. Lehner ([30][31]) has provided empirical evidence that the activities of ‘maintenance’ change during the lifecycle of a project. However, other than this, very little data has been collected, and our evidence is gleaned from published case studies and personal practical experience. The experience from these projects is summarized in the rest of this section.

### **6.1 The Microsoft Corporation**

The description of the Microsoft development process, as given by Cusumano and Selby [42], illustrates the techniques and processes used for high volume mass market shrink-wrapped software. In particular, we can draw on the following evidence:

1. At Microsoft, there is a substantial investment in the initial development stage, before revenue is generated from sales. This includes testing.
2. The division between initial development and evolution is not sharp; the technique of using beta releases to gain experience from customers is widely used.
3. Microsoft tries to avoid quite explicitly the traditional maintenance phase. It is realized that with such a large user base, this is logistically impossible. Object code patches (service packs) are released to fix serious errors, but not for feature enhancement.

4. The development of the next release is happening while the existing release is still achieving major market success. Thus Windows 98 was developed while Windows 95 was still on its rising curve of sales. Microsoft did not wait until Windows 95 sales start to decline to start development, and to do so would have been disastrous. Market strategy is based upon a rich (and expanding) set of features. As soon as Windows 98 reached the market, sales of Windows 95 declined very rapidly. Shortcomings and problems in Windows 95 were taken forward for rectification in Windows 98; they were not addressed by maintenance of Windows 95.
5. Microsoft does not support old versions of software, which have been phased out, but they do provide transition routes from old to new versions.
6. Organizational learning is becoming evident through the use of shared software components. Interestingly, Microsoft has not felt the need for substantial documentation, indicating that the tacit knowledge is retained effectively in design teams.

We conclude that evolution represents Microsoft's main activity, and servicing by choice a very minor activity.

## **6.2 The VME operating system**

This system has been implemented on ICL (and other) machines for the past 30 years or so, and has been written up by Holt [41]. It tends to follow the

classical X.Y release form, where X represents a major release (evolution) and Y represents minor changes (servicing). In a similar way to Microsoft, major releases tend to represent market-led developments incorporating new or better facilities.

The remarkable property of VME is the way in which its original architectural attributes have remained over such a long period, despite the huge evolution in the facilities. It is likely that none of the original source code from the early 1970's still is present in the current version. Yet its architectural integrity is clearly preserved. We can deduce:

1. There was a heavy investment in initial development, which has had the effect of a meticulous architectural design. The system has been evolved by experts with many years of experience, but who also have been able to sustain architectural integrity.
2. Each major release is subject to servicing, and eventually that release is phased-out and closed down.
3. Reverse engineering is not used from one major release to another; evolution is accomplished by team expertise and an excellent architecture.

### **6.3 The Y2K experience**

An excellent example of software in its servicing stage and its impact has been provided by the "Y2K" episode. It was caused by a widespread

convention that limited the representation of a year in a date to the last two digits, for example the year 1997 was represented by two digits “97”. Based on this convention, when January 1, 2000 was reached, the year represented as “00” would be interpreted by computers as 1900, with all accompanying problems such misrepresentation could cause.

The origin of the two-digit convention goes back to the early programs of the 1950s when the memory space was at a premium and hence to abbreviate the year to its two final digits seemed reasonable, while the problems this would cause seemed very remote. Even as the time was moving closer towards the fateful January 1, 2000, programmers continued to use the entrenched convention, perhaps out of inertia and habit.

The extent of the problem became obvious in late 1990s and a feverish attempt to remedy the problem became widespread. Articles in the popular press and by pessimists predicted that the programs would not be updated on time, that the impacts of this failure would be catastrophic, disrupting power supplies, goods distribution, financial markets, etc. At the height of the panic, the president of the USA appointed a “Y2K czar” with an office close to the White House whose role was to coordinate the efforts to fix Y2K problem (and if this did not succeed, to deal with the ensuing chaos). Similar appointments were made in other countries, such as the UK.

Fortunately, the dire Y2K predictions did not materialize. Many old programs were closed down. The programs that could not be closed down

and needed to be repaired were indeed fixed, mostly by a technique called “windowing” which is a variant of wrapping. The two-digit dates are re-interpreted by moving the “window” from years 1900 – 1999 to a different period, for example 1980 – 2080. In the new window, “99” is still interpreted as 1999, but “00”, “01”, ... , are now interpreted as 2000, 2001, etc. This worked well (for the time being) and has postponed the problem to the time when the new “window” will run out. The Y2K czar quietly closed his office and left town. There were, in fact, very few reported problems.

However the whole Y2K effort is estimated to have had a staggering cost. It is estimated that world-wide, about 45% of all applications were modified and 20% were closed down, at the cost between \$375 and \$750 billion [55].

From the viewpoint of our model, the Y2K problem was caused by the fact that many legacy systems were in the servicing stage, and although Y2K rectification would be only a routine change during the evolution stage, it was a hard or very hard change during the servicing stage. At heart, the problem was caused by a design decision (a key data representation choice) and changes in design are very hard to achieve successfully during servicing. The reason why the Y2K problem caught so many managers by surprise is the fact that the difference between evolutionary and servicing stages was not well understood.

#### **6.4 A major billing system**

This 20 year old system generates revenue for its organization, and is of strategic importance. However, the marketplace for the organization's products has changed rapidly in recent years, and the billing system can no longer keep up with market-led initiatives (such as new products). Analysis shows that this system has slid from evolution into servicing without management realizing it; the key designers have left; the architectural integrity has been lost; changes take far too long to implement, and revalidation is a nightmare; it is a classical legacy system. The only solution (at huge expense) is to replace it.

### **6.5 A small security company**

A small company has a niche market in specialized hardware security devices. The embedded software is based around Microsoft's latest products. The products must use rapidly changing hardware peripherals, and the company must work hard to keep ahead of the competition in terms of the sophistication of the product line.

The software therefore consists of COTS components (e.g. special device drivers), locally written components, some legacy code, and glue written in a variety of languages (e.g. C, C++, BASIC). The system was not planned in this way, but has evolved into this form because of 'happenstance'.

The software is the source of major problems. New components are bought, and have to work with the old legacy. Powerful components are linked via

very low-level code. Support of locally written components is proving very hard.

From our perspective, we have a software system in which some parts are in initial development, some are in evolution, others are in servicing, while others are ready for phase-out. There is no sustained architectural design.

The existing (Lientz and Swanson) type analysis sheds little light on this problem. Our model allows each component and connector to be assessed in terms of its stage. This should then allow the company to develop a support plan. For example, a component that is being serviced can have a dependency on another component that is being evolved.

## **6.6 A long-lived defense system**

A different type of case study is represented by a long-lived, embedded, defense system which is safety related. This was developed initially many years ago (in Assembler) and needs to be continually updated to reflect changes in the supporting hardware. In classic terms, this system would be thought of as being in the maintenance phase, but according to our analysis, it is still being evolved, yet surely but inexorably slipping into servicing:

1. The software is still core to the organization, and will be for many years.  
Failure of the software in service would be a disaster.

2. Many experts with in-depth knowledge of the software (and hardware) are working on the system. They understand the architecture and are Assembler experts. The software is being changed to meet quite radical new requirements. It is free from ad-hoc patches, and consistent documentation is being produced though structurally it is decaying. Comprehensive test procedures are in place and are used rigorously. The system engineers understand the impact of local changes on global behavior. Mature, well-understood processes are employed.
3. Conversely, some experts have recently left the organization, and this loss of expertise, accompanied by structural decay mentioned above, is a symptom of serious software decay. Reverse engineering is not considered feasible, partly because of the lack of key expertise. If the process of the decay reaches a certain point, it is likely that the system will be developed again *ab initio*.

### **6.7 A printed circuits program**

One of the co-authors was for several years a software manager of a software department. One of the projects in that department was the program that designed printed circuit boards, and was used by the customers within the same institution. Because of the critical nature of the product, it had to be constantly evolved as new requirements appeared and had to be satisfied. The original developers were evolving the program but at the same

time, they were some of the most qualified program developers in the department.

Since there was a backlog of other projects that required high expertise, and the difference between evolution and servicing was not understood at the time, the manager tried several times unsuccessfully to transfer the evolution responsibility to different people. However all attempts to train new programmers so that they would be able to take over the evolution task and relieve the original developers turned out to be unsuccessful. In all instances, the new trainees were able to do only very limited tasks and were unable to make strategic changes in the program. At that time, this inability to transfer a "maintenance" task proved to be baffling to the manager. In hindsight, the expertise needed for evolution was equivalent or perhaps even greater than the expertise to create the whole program from scratch. It proved more cost effective to assign the new programmers to the new projects and to leave the experienced developers to evolve the printed circuit program.

## **6.8 Project PET**

This case study is an example of an attempted reengineering project.

PET is a CAD tool developed a car company [56][57] to support the design of the mechanical components (transmission, engine etc.) of a car. It is implemented in C++ and every mechanical component is modeled as a C++

class. The mechanical component dependency is described by a set of equations that constitute a complex dependency network. Whenever a parameter value is changed, an inference algorithm traverses the entire network and recalculates the values of all dependent parameters. PET consists of 120 000 lines of C++ code and is interfaced with other CAD software, including 3-D modeling software.

After the initial implementation, there was a massive stage of evolution where in our estimate, more than 70% of the current functionality was either radically changed or newly introduced. The evolution was driven mostly by the user requests. All changes to PET were performed as quickly as possible in order to make the new functionality available. This situation prevented conceptual changes to the architecture, and the architecture progressively deteriorated. Also the original architecture was not conceived for changes of this magnitude. As a result, the architecture drastically deteriorated to the point where the requested evolutionary changes are becoming increasingly difficult. The symptoms of deterioration include the introduction of clones into the code and misplacement of code into the wrong classes. During a code review we identified 10% of the PET code as clones.

Because of code deterioration, the evolvability of the PET software has been decreasing and some evolutionary changes are becoming very hard. An example of a hard change is a modification to the inferencing algorithms. As mentioned above, the program uses inferencing by which the relationships between the mechanical components are maintained. The program would

greatly benefit from an introduction of a commercially available component for inferencing that contains more powerful inferencing algorithms, but the current architecture with misplaced code and clones does not make that change feasible. Because of this, the changes done to the software have the character of patches that further corrode the architecture.

Recently a decision was made to move PET software into a servicing stage, with work performed by a different group of people, and to stop all evolutionary changes. While PET will be serviced and should meet the needs of the users in this situation, a new version of PET will be developed from scratch, embodying all the expertise gained from the old PET evolution. The attempt to re-engineer the old version of PET has been abandoned.

## **6.9 The FASTGEN geometric modeling toolkit**

FASTGEN is a collection of Fortran programs used by the U. S. Department of Defense to model the interactions between weapons (such as bombs or missiles) and targets (such as tanks, or airplanes). Targets are modeled as large collections of triangles, spheres, donuts and other geometric figures, and ray tracing programs compute the effects of the explosion of a weapon.

FASTGEN was originally developed in the late 70's by one contractor, and has since been modified many times by other agencies and contractors at different sites ranging from California to Florida. Originally developed

primarily for mainframe computers, it has been ported to supercomputer platforms such as CDC and Cray, Digital Equipment VAX, and, in the 90's to PC and Unix workstations.

A study of CONVERT, one of the FASTGEN programs, illustrates the impact of the original architecture on program comprehension and evolvability [58]. The original code was poorly modularized with large, non-cohesive subroutines and heavy use of global data. The program still contains several optimizations that were important for the original mainframe environment, but that now make comprehension very difficult. For example, records are read and written in arbitrary batches of 200 at a time; in the original environment input/output could cause the program to be swapped out of memory so it was much more efficient to read many records before doing computations. Current versions of the program preserve this complex batching logic that is now obscure and irrelevant.

FASTGEN is now in a late servicing stage, bordering on phase-out.

### **6.10 A financial management application**

This application dates from the 1970s, when it was implemented on DEC PDP computers. Recently it has been ported to PC/Windows machines. It is financially critical to its users. The software is modest in size (around 10,000 lines of code). Prior to the port, the software was stable and had

evolved very little. In Lehman's terms this was an S type system, with very little evolution, and very long lived.

During the port, it had been decided to modify the code, preserving the original architecture as far as possible. Unfortunately, this had the following effects:

- a) On the PDP series, different peripheral drivers (magnetic tapes, paper tape, discs, etc.) had very different interfaces. These differences were not well hidden, and impacted much of the application code. In the PC implementation, Windows has a much cleaner unified view of access to discs CD's etc. (i.e. byte vectors). Yet the original PDP peripheral code structure was retained, because the designers of the port could not be sure of correctly handling all side effects if the structure were changed. As a result, the code is much longer than it needs to be, with much redundancy and unwarranted complexity.
- b) Even worse, the application needs to run in real time. The real time model employed in the original language has been retained in the port, yet the model for the new application language has been added. The result is a labyrinthine real time program structure that is extremely hard to comprehend.

This application has now slipped to the end of the servicing stage and only the simplest changes are possible. The expertise does not exist to re-

engineer it. If major changes are needed, the system will have to be rewritten.

## **7. Software Change and Comprehension**

### **7.1 The miniprocess of change**

During both the evolution and the servicing stages, a software system goes through a series of changes. In fact, both evolution and servicing consist of repeated change, and hence understanding the process of software change is the key to understanding these stages and the problems of the whole software lifecycle. Accordingly in this section we look at the process of change in more detail, decomposing change into its constituent tasks. A particularly important task is program comprehension, because it consumes most of the programmer's time, and its success dominates what can or cannot be accomplished by software change.

The tasks comprising software change are listed in [14] (see the Introduction). They are summarized in the *miniprocess of change*. In order to emphasize tasks that we consider important, we divide them differently than the standard and group them into the miniprocess in the following way:

- Change request: the new requirements for the system are proposed.
- Change planning: to analyze the proposed changes.
  - Program comprehension: understand the target system.

- Change impact analysis: analyze the potential change and its scope.
- Change implementation: the change is made and verified
  - Restructuring (re-factoring) for change.
  - Initial change.
  - Change propagation: make secondary changes to keep the entire system consistent.
  - Validation and verification: to ensure if the system after change meets the new requirement and that the old requirements have not been adversely impacted by the change.
  - Re-documentation: to project the change into all documentation.
- Delivery

These tasks are discussed in more detail in this section.

## **7.2 Change request and planning**

The users of the system usually originate the change requests (or maintenance requests). These requests have the form of fault reports or requests for enhancements. Standard practice is to have a file of requests (backlog) that is regularly updated. There is a submission deadline for change requests for the next release. After the deadline, the managers decide which particular requests will be implemented in that release. All requests

that are submitted after the deadline or the requests that did not make it into the release will have to wait for the following release.

Even this superficial processing of change requests requires some understanding of the current system so that the effort required may be estimated. It is a common error to underestimate drastically the time required for a software change and thus the time to produce a release.

In small changes, it suffices to find the appropriate location in the code and replace the old functionality with the new one. However large incremental changes require implementation of new domain concepts. Consider a retail "point-of-sale" application for handling bar code scanning and customer check-out. The application would need to deal with several forms of payment, such as cash and credit cards. An enhancement to handle check payments would involve a new concept, related to the existing payment methods but sufficiently different to require additional data structures, processing for authorization, etc. There will be quite a lot of new code, and care is needed to maintain consistency with existing code to avoid degrading the system architecture.

Concepts that are dependent on each other must be implemented in the order of their dependency. For example, the concept "tax" is dependent on the concept "item" because different items may have different tax rates and tax without an item is meaningless. Therefore, the implementation of "item"

must precede the implementation of “tax”. If several concepts are mutually dependent, they must be implemented in the same incremental change.

Mutually independent concepts can be introduced in arbitrary order, but it is advisable to introduce them in the order of importance to the user. For example, in the point-of-sale program it is more important to deal correctly with taxes than to support several cashiers. An application with correct support for taxes is already usable in stores with one cashier. The opposite order of incremental changes would postpone the usability of the program.

Change planning thus requires the selection of domain concepts to be introduced or further developed. It also requires finding in the old code the location where these concepts should be implemented so that they properly interact with the other already present concepts. Obviously these tasks require a deep understanding of the software and of its problem domain,

### **7.3 Change implementation**

Implementation of the software change requires several tasks, often with some looping and repetition. If the proposed change has a large impact on the architecture, there may be a preliminary restructuring of the program to maintain cleanness of design. In an object oriented program, for example, this may involve re-factoring to move data or functions from one class to another [59].

The actual change may occur in any one of several ways. For small changes, obsolete code is replaced by a new code. For large incremental changes, new code is written and then “plugged” into the existing system. Several new classes implementing a new concept may be written, tested and interfaced with the old classes already in the code.

Very often the change will propagate, that is, it will require secondary changes. In order to explain change propagation, we have to understand that software consists of *entities* (classes, objects, functions, etc.) and their *dependencies*. A dependency between entities A and B means that entity B provides certain services, which A requires. A function call is an example of a dependency among functions. Different programming languages or operating systems may provide different kinds of entities and dependencies. A dependency of A on B is *consistent* if the requirements of A are satisfied by what B provides.

Dependencies can be subtle and of many kinds. The effect may be at the code level; for example, a module under change may use a global variable in a new way, so all uses of the global variable must be analyzed (and so on). Dependencies can also occur via non-functional requirements or business rules. For example in a real time system, alteration of code may affect the timing properties in subtle ways. For this reason, the analysis of a change, and the determination of which code to alter often cannot easily be compartmentalized. Senior maintenance engineers need a deep understanding of the whole system and how it interacts with its environment

to determine how a required change should be implemented while hopefully avoiding damage to the system architecture. The business rules may be extremely complex (e.g. the “business rules” that address the navigation and flight systems in an on-board safety critical flight control system); in an old system, any documentation on such rules has probably been lost, and determining the rules retrospectively can be an extremely time-consuming and expensive task (for example, when the domain expert is no longer available).

Implementation of a change in software thus starts with a change to a specific entity of the software. After the change, the entity may no longer fit with the other entities of the software, because it may no longer provide what the other entities require, or it may now require different services from the entities it depends on. The dependencies that no longer satisfy the require-provide relationships are called inconsistent dependencies (inconsistencies for short), and they may arise whenever a change is made in the software. In order to reintroduce consistency into software, the programmer keeps track of the inconsistencies and the locations where the secondary changes are to be made. The secondary changes, however, may introduce new inconsistencies, etc. The process in which the change spreads through the software is sometimes called the ripple effect of the change [60][61]. The programmer must guarantee that the change is correctly propagated, and that no inconsistency is left in the software. An unforeseen and uncorrected inconsistency is one of the most common sources of errors in modified software.

A software system consists not just of code, but also of documentation. Requirements, designs, test plans, and user manuals can be quite extensive and they often are also made inconsistent by the change. If the documentation is to be useful in the future, it must also be updated.

Obviously the modified software system needs to be validated and verified. The most commonly used technique is regression testing, in which a set of system tests is conserved and rerun on the modified system. The regression test set needs to have fairly good coverage of the existing system if it is to be effective. It grows over time as tests are added for each new concept and feature added. The regression test set will also need to be rerun many times over the life of the project. Regression testing is thus not cheap, so it is highly desirable to automate the running of the tests and the checking of the output.

Testing is, however, not enough to guarantee that consistency has been maintained. Inspections can be used at several points in the change miniprocess to confirm that the change is being introduced at the right point, that the resulting code meets standards, and that documentation has indeed been updated for consistency.

It is evident that clear understanding of the software system is essential at all points during change implementation. Refactoring requires a vision of the architecture and of the division of responsibilities between modules or

classes. Change propagation analysis requires tracing the dependencies of one entity on another, and may require knowledge of subtle timing or business rule dependencies. Documentation updating can be among the most knowledge-demanding tasks since it requires an awareness of the multiple places in each document where any particular concept is described. Even testing requires an understanding of the test set, its coverage, and of where different concepts are tested.

In the previous paragraphs we have described what *should* be done as part of each software change. Given the effort required to "do it right" it is not surprising to discover that, in practice, some of these tasks are skipped or slighted. In each such case, a trade-off is being made between the immediate cost or time and an eventual long-term benefit. As we will discuss in section 8 it is not necessarily irrational to choose the immediate over the long-term, but all such decisions need to be taken with full awareness of the potential consequences.

As we have described earlier, program comprehension is a key determinant of the lifecycle of any specific software product. To understand why this is so, we need to understand what it means to understand a program, why that understanding is difficult, and how it fits into the cycle of software change.

#### **7.4 Program comprehension**

Program comprehension is carried out with the aim of understanding source code, documentation, test suite, design, etc. by human engineers. It is typically a gradual process of building up understanding, which can then be used further to explain the construction and operation of the program. So program comprehension is the activity of understanding of how a program is constructed and its underlying intent. The engineer requires precise knowledge of the data items in the program, the way these items are created, and their relationships [62].

Various surveys have shown that the central activity in maintenance is understanding the source code. Chapin and Lau [63] describe program comprehension as the most skilled and labor intensive part of software maintenance, while Oman [64] states that the key to effective software maintenance is program comprehension. Thus it is a human-intensive activity that consumes considerable costs. An early survey to the field is [13]; see also von Mayrhauser [65]. The understanding can then be used for:

- Maintenance and evolution (e.g. [66])
- Reverse engineering (e.g. [13])
- Learning and training
- Redocumentation (e.g. [67][68])
- Reuse (e.g. [69])
- Testing and validation (e.g. [70])

The field has prompted several theories derived from empirical investigation of the behavior of programmers. There are three fundamental views, see Storey [71]:

- Comprehension is undertaken in a top-down way, from requirements to implementation [72][73]
- Comprehension is undertaken in a bottom up way, starting with the source code, and deducing what it does and how it does it [74]
- Comprehension is undertaken opportunistically [75][76]

All three may be used at different times, even by a single engineer. It is encouraging to note that much work on comprehension has been supported by empirical work to gain understanding of what engineers actually do in practice (see for example, [66][77][76]). To support comprehension, a range of tools has been produced and some of these present information about the program, such as variable usage, call graphs etc. in a diagrammatic or graphical form. Tools divide into two types:

- Static analysis tools, which provide information to the engineer based only on the source code (and perhaps documentation)
- Dynamic analysis tools, which provide information as the program executes.

More recent work is using virtual reality and much more sophisticated visualization metaphors to help understanding [78].

The work on an integrated metamodel [65] has drawn together into a single framework the work on cognition of large software systems. It is based on four components:

- Top-down structures
- Situation model
- Program model
- The knowledge base.

It combines the top down perspective with the bottom up approach (i.e. situation and program models). The knowledge base addresses information concerned with the comprehension task, and is incremented as new and inferred knowledge is determined. The model is not prescriptive, and different approaches to comprehension may be invoked during the comprehension activity.

All authors agree that program comprehension is a human-oriented and time intensive process, requiring expertise in the programming language and environment, deep understanding of the specific code and its interactions, and also knowledge of the problem domain, the tasks the software performs in that domain, and the relationships between those tasks and the software structure.

As mentioned earlier, locating concepts in the code is a program comprehension task that is very important during the phase of change planning. Change requests are very often formulated as requests to change

or introduce implementation of specific domain concepts, and the very first task is to find where these concepts are found in the code. A usual assumption behind the concept location task is that the user does not have to understand the whole program, but only the part that is relevant to the concepts involved in the change.

In a widely cited paper, Biggerstaff [79] et al. presented a technique of concept location in the program based on the similarity of identifiers used in the program and the names of the domain concepts. When trying to locate a concept in the code, the programmer looks for the variables, functions, classes, etc. with a name similar to the name of the concept. For example when trying to locate implementation of breakpoints in a debugger, the programmer looks for variables with identifiers `breakPoint`, `BreakPoint`, `break_point`, `brkpt`, etc. Text pattern matching tools like “grep” are used for this purpose. Once the appropriate identifier is found, the programmer reads and comprehends the surrounding code in order to locate all code related to the concept being searched.

Another technique of concept or feature location is based on analysis of program execution traces [80]. The technique requires instrumentation of the program so that it can be determined which program branches were executed for a given set of input data. Then the program is executed for two sets of data: data set A with the feature and data set B without the feature. The feature is most probably located in the branches that were executed for data set A but were not executed for data set B.

Another method of concept location is based on static search of code [81]. The search typically starts in function `main()` and the programmer tries to find the implementation of the concept there. If it cannot be located there, it must be implemented in one of the sub-functions called from `main()`, hence the programmer decides which sub-function is the most likely to implement the concept. This process is recursively repeated (with possible backtracks) until the concept is found.

As remarked earlier, we believe that the comprehensibility of a program is a key part of software quality and evolvability, and that research in program comprehension is one of the key frontiers of research in software evolution and maintenance.

## **8. Sustaining Software Value**

### **8.1 Staving off the servicing stage**

One of the goals of our staged model of the software lifecycle is to aid software managers in thinking about the systems they control and in planning their futures. It is clear from our argument so far that a software system subtly loses much of its value to its owners when it makes the transition from the evolution to the servicing stage.

A software system in the evolution stage is routinely adapted to changing organizational needs and can thus make a considerable contribution to the organization's mission and/or revenues. When the system transitions into servicing, only the simplest changes can be made; the software is a less valuable asset and may actually become a constraint on the organization's success.

Thus most software managers will want to stave off the servicing stage as long as possible. There are a number of strategies that can be adopted to sustain software value, but unfortunately all of them produce their benefits in the long term, but require an expenditure of effort or time in the short term. A software manager must seek an appropriate trade-off between the immediate budget and time pressures of doing business and the potential long-term benefits of increased software value. The appropriate choice of strategies will obviously not be the same for all companies. An e-business that must change almost daily to survive will focus on rapid change, whereas the owners of an embedded system with stable requirements but life-critical consequences of failure may be able to focus on long-term quality.

Thus this section is not prescriptive, but merely tries to identify some of the issues that a software manager or chief architect should consider. We list some of the strategies and techniques that have been proposed, categorizing them by their stage in the lifecycle. Unfortunately there seems to be very little published analysis that would aid a software manager in estimating the

costs and benefits. Research into the actual effectiveness of each would seem to be a priority.

## 8.2 Strategies during development

The key decisions during development are those that determine the architecture of the new system and the team composition. These decisions are, of course interrelated; as has been mentioned, many of the more famously evolvable systems such as Unix and VME were the product of a very few highly talented individuals. Advice to "hire a genius" is not misplaced, but is difficult to follow in practice.

In the current state of the art, there is probably little that can be done to design an architecture to permit any conceivable change. However it is possible to address systematically those potential changes that can be anticipated, at least in general terms. For instance it is well known that changes are extremely common in the user interfaces to systems, to operating systems, and to hardware, while the underlying data and algorithms may be relatively stable. During initial development a roughly prioritized list of the anticipated changes can be a very useful guide to architectural design [82].

Once possible changes are identified, the main architectural strategy to use is *information hiding* of those components or constructs most likely to change. Software modules are structured so that design decisions, such as

the choice of a particular kind of user interface or a specific operating system, are concealed within one small part of the total system, a technique described by Parnas since the early 70's [83]. If the anticipated change becomes necessary in the future, only a few modules would need to be modified.

The emergence of object oriented languages in the 90's has provided additional mechanisms for designing to cope with anticipated changes. These languages provide facilities such as abstract classes and interfaces which can be subclassed to provide new kinds of object which are then used by the rest of the program without modification. Designers can also make use of object oriented design patterns many of which are intended to provide flexibility to allow for future software enhancements [84]. For example the Abstract Factory pattern provides a scheme for constructing a family of related objects that interact, such as in a user interface toolkit. The pattern shows how new object classes can be added, say to provide an alternate look-and-feel, with minimal change to the existing code.

In both traditional and object oriented systems, architectural clarity and consistency can greatly facilitate program comprehension. Similar features should always be implemented in the same way, and if possible by similarly named components. For example, one of the authors once studied a test coverage monitoring system that provided coverage of basic blocks, of decisions, and of several kinds of data flows [85]. The code for each kind of monitoring was carefully segregated into source files, with code to display

blocks in a file called `bdisp.c`, that to display decisions in `ddisp.c`, and so on. Within each file, corresponding functions were named the same, for example each file had a `display()` function to handle its particular kind of display. This consistency made it very easy for a maintainer to hypothesize where particular concepts were likely to be located, and greatly speeded up understanding of the program architecture, even in the absence of design documentation.

On the other hand, inconsistency in design and naming can lead future maintainers into errors in understanding. Polymorphic object oriented systems are particularly susceptible to this kind of error, since many member functions may have the same name. They thus cannot easily be distinguished by a maintainer reading the code. If they do not perform precisely the same task, or if one version has side effects not shared by others, then the maintainer may be seriously misled in reading the code [86].

The best way to achieve architectural consistency is probably to leave the basic design in the hands of a very small team who work together very closely. Temptations to rotate personnel between projects should probably be strongly resisted at this stage. Other programmers may later maintain the consistent design if they are encouraged to study the structure of existing code before adding their own contributions. Consistency may then further be enforced by code inspections or walkthroughs.

If the project architecture depends on purchased commercial off the shelf components (COTS), as so many modern projects do, then particular care is needed. First, it would obviously be dangerous to depend heavily on a component that is already in the servicing or phase-out stages. It is thus important to understand the true status of each component, which may require information that the component supplier is reluctant to give. Second, the impact of the possible changes on the COTS component should be considered. For instance, if changes to the hardware platform are anticipated, will the COTS supplier, at reasonable cost, evolve his product to use the new hardware? If not, information hiding design might again be advisable to facilitate the possible substitution of a different COTS component in the future.

Programming environments and software tools that generate code may create problems similar to those of COTS. Many such environments assume implicitly that all future changes will take place within the environment. The generated code may be incomprehensible for all practical purposes. Unfortunately experience indicates that environments, and the companies that produce them, often have much shorter lifetimes than the systems developed using them.

Finally, there are well-know coding practices that can greatly facilitate program comprehension and thus software change. In section 2.4 we have mentioned the use of IEEE or ISO standards, the enforcement of house

coding style to guarantee uniform layout and commenting and an appropriate level of documentation to match the criticality of the project.

One coding technique that should probably be used more often is to insert instrumentation into the program to aid in debugging and future program comprehension. Experienced programmers have used such instrumentation for years to record key events, inter-process messages, etc. Unfortunately instrumentation is rarely mandated, and more often introduced ad-hoc only after a project has fallen into trouble [87]. If used systematically, it can be a great aid to understand the design of a complex system "as-built" [88].

We should mention again that all the above techniques involve a trade-off between evolvability and development time. The study of potential changes takes time and analysis. Design to accommodate change may require more complex code, which impacts both time and later program comprehension. (At least one project found it desirable to remove design patterns that had been introduced to provide unused flexibility [89]). COTS components and programming environments can greatly speed up initial development, but with serious consequences for future evolvability. Design consistency and coding standards are difficult to enforce unless almost all code is inspected, a time-consuming and thus far from universal practice. In the rush to get a product to market, a manager must be careful about decisions that sacrifice precious time against future benefit.

### 8.3 Strategies during evolution

During the evolution phase, the goal must be to delay servicing as long as possible by preserving a clean architecture and by facilitating program comprehension.

As previously mentioned, most modern projects transition to the evolution phase with at least some key members of the original development team in place. There are, of course, circumstances when this may not be possible and a transition to a new team is unavoidable. This is a strategy which risks an almost immediate slide into servicing due to the difficulties of program comprehension. If a new team is, however, absolutely essential, then there are some steps that can be taken, such as having developers on-site for some months to aid the transition. Software managers placed in this situation may want to consult some of the papers by Pigoski and others that discuss experiences in software transition [90][91][92].

If it has not already been done, the system should now be placed under configuration management control. From here on different customers will have different versions, and it is essential to have a mechanism for tracking the revisions of each file that went into making up each version. Without such a mechanism it will be very difficult to interpret problem reports arriving from the field. Change control may be formal or informal depending on the process used, but change control procedures should be well defined, and it is highly desirable to have one person designated as

configuration manager, with responsibility for change control and version tracking.

At this stage, if not earlier, the project needs to have a clear strategy for program comprehension. That strategy can include combinations of at least three elements:

- Team knowledge, carried by team members who commit to long-term participation in the project
- Written documentation of specifications, design, configurations, tests, etc., or its equivalent in the repository of a software tool
- Reverse engineering tools to recover design information from the system itself.

As evolution starts, team composition may change somewhat, often with some shrinkage. If the project manager intends to rely mainly on team knowledge for program comprehension, this is a good point to take inventory of the available knowledge and to try to avoid over concentration in one or two team members. As previously mentioned, agile development methods such as Extreme Programming often use pair programming, in which two programmers work together on each task [40]. As indicated by the printed circuits program case described in section 6.7 it is probably unrealistic to expect new programmers to undertake major changes alone, but it may be possible to get them to work in a pair with a more experienced programmer.

It is desirable to avoid the well-known "guru" phenomenon, in which one person is the only expert in an important part of the project. The guru often tends to adopt that part as his territory, and makes it difficult for anyone else to work with that code. Once this situation is established, it can be very difficult to manage. A guru problem is obviously dangerous for the future of the project and can also have a bad impact on team moral.

If the manager decides to emphasize written documentation for program comprehension, then there will be considerable overhead to revise and update all such documentation as the system evolves. It is likely that any design documentation available at the beginning of evolution will represent the system the developers intended to build, which may differ substantially from what was actually built.

A useful technique is incremental re-documentation, in which documentation is generated or updated as modules are modified. A trainee programmer may be assigned to do the write up, based on notes or an interview with the experienced programmer who actually made the modification, thus reducing the cost and schedule impact of the re-documentation. One of the authors has given a case study showing how this strategy was applied to a large C++ system [93].

The manager needs to establish a tricky balance between programmers' desire to restructure or rewrite code and the economics of the project. Unless they wrote it themselves, programmers will almost always complain

of the quality of the code they encounter! Some such complaints are certainly well founded as restructuring and refactoring may be needed to maintain a clean architecture. However if all such requests are approved, the burden of coding, inspecting, and especially testing, will become unsustainable.

Key decisions in the evolution phase concern the creation of new versions and the transition to servicing. If the versioned staged model of Figure 2 is followed, the new version will probably start development well before the old one passes into servicing. Management should make a conscious decision as to the paths to be followed, based on judgments about the state of the system and the demands of the market.

#### **8.4 Strategies during servicing**

The transition into servicing implies that further change will be relatively minor, perhaps involving bug fixes and peripheral new features. It is important to understand that the transition is largely irreversible since essential knowledge and architectural integrity have probably been lost. If the product is to be reengineered, it is likely that the best strategy will be simply to try to reproduce its black-box behavior rather than to study and reuse its current code or design [94].

Often there may be a transition to a new maintenance team as servicing begins. Expectations about what can be accomplished by such a team should

be kept modest to avoid impossible commitments. Configuration management must continue in place to be able to understand reports from the field. Strategies such as opportunistic re-documentation may still be desirable, but fixes that degrade the code may be tolerated since the basic strategy is to minimize cost while maintaining revenue in the short run.

Finally, the servicing stage is the time to make and implement a plan for phase-out. Migration paths to a new version in development should be provided. The main issue is often the need to recover and reformat vital organizational data so that it can be used in the new version.

## **9. Future Directions: Ultra rapid software evolution**

It is possible to inspect each activity of the staged software model and determine how it may be speeded up. Certainly, new technology to automate parts may be expected, supported by tools (for example, in program comprehension, testing etc.). However, it is very difficult to see that such improvements will lead to a *radical* reduction in the time to evolve a large software system. This prompted us to believe that a new and different way is needed to achieve “ultra rapid evolution”; we term this “evolution in internet time”. It is important to stress that such ultra rapid evolution does not imply poor quality, or software that is simply hacked together without thought. The real challenge is to achieve very fast change yet provide very high quality software. Strategically, we plan to achieve this by bringing the evolution process much closer to the business process. The generic problem

of ultra rapid evolution is seen as one of the grand challenges for software engineering (see [95][96]).

The staged model allows us to address a large system built out of many parts (and so on recursively). Each part may be in one of the five stages (though we would expect the main stress to be the first three stages). This has been ignored in previous research. The integration mechanism is market-led, not simply a technical binding, and requires the representation of non-technical and non-functional attributes of the parts. The new perspective offered by the staged model has been a crucial step in developing a *serviceware* approach.

For software evolution, it is useful to categorize contributing factors into those which can rapidly evolve, and those which cannot, see Table 1.

<Table 1 near here>

We concluded that a “silver bullet”, which would somehow transform software into something which could be changed (or could change itself) far more quickly than at present, was not viable. Instead, we take the view that software is actually hard to change, and thus that change takes time to accomplish. We needed to look for other solutions.

Let us now consider a very different scenario. We assume that our software is structured into a large number of small components which exactly meet the user’s needs and no more. Suppose now that a user requires an improved component C. The traditional approach would be to raise a change request

with the vendor of the software, and wait for several months for this to be (possibly) implemented, and the modified component integrated.

In our solution, the user disengages component C, and searches the marketplace for a replacement C' which meets the new needs. When this is found, it is bound in instead of C, and used in the execution of the application. Of course, this assumes that the marketplace can provide the desired component. However, it is a well established property of marketplaces that they can spot trends, and make new products available when they are needed. The rewards for doing so are very strong and the penalties for not doing so are severe. Note that any particular component supplier can (and probably will) use traditional software maintenance techniques to evolve their components. The new dimension is that they must work within a demand-led marketplace. Therefore, if we can find ways to disengage an existing component and bind in a new one (with enhanced functionality and other attributes) ultra rapidly, we have the potential to achieve ultra-rapid evolution in the target system.

This concept led us to conclude that the fundamental problem with slow evolution was a result of software that is marketed as a product, in a supply-led marketplace. By removing the concept of ownership, we have instead a service i.e. something that is used, not owned. Thus we generalized the component-based solution to the much more generic service based software in a demand led marketplace [97].

This service-based model of software is one in which services are configured to meet a specific set of requirements at a point in time, executed and disengaged - the vision of instant service, conforming to the widely accepted definition of a service:

*“an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production” [98].*

Services are composed out of smaller ones (and so on recursively), procured and paid for on demand. A service is not a mechanized process; it involves humans managing supplier-consumer relationships. This is a radically new industry model, which could function within markets ranging from a genuine open market (requiring software functional equivalence) to a *keisetsu* market, where there is only one supplier and consumer, both working together with access to each other's information systems to optimize the service to each other.

This strategy potentially enables users to create, compose and assemble a service by bringing together a number of suppliers to meet needs at a specific point in time. An analogy is selling cars: today manufacturers do not sell cars from a pre-manufactured stock with given color schemes, features etc.; instead customers configure their desired car from series of options and only then is the final product assembled. This is only possible

because the technology of production has advanced to a state where assembly of the final car can be undertaken sufficiently quickly.

Software vendors attempt to offer a similar model of provision by offering products with a series of configurable options. However this offers extremely limited flexibility -- consumers are not free to substitute functions with those from another supplier since the software is subject to *binding* which configures and links the component parts, making it very difficult to perform substitution. The aim of this research is to develop the technology which will enable *binding* to be delayed until immediately before the point of execution of a system. This will enable consumers to select the most appropriate combination of services required at any point in time.

However *late binding* comes at a price, and for many consumers, issues of reliability, security, cost and convenience may mean that they prefer to enter into contractual agreements to have *some early binding* for critical or stable parts of a system, leaving more volatile functions to late binding and thereby maximizing competitive advantage. The consequence is that any future approach to software development must be interdisciplinary so that non-technical issues, such as supply contracts, terms and conditions, and error recovery are addressed and built in to the new technology.

A truly service-based role for software is far more radical than current approaches, in that it seeks to change the very nature of software. To meet users' needs of evolution, flexibility and personalization, an open market-

place framework is necessary in which the most appropriate versions of software products come together, are bound and executed as and when needed. At the extreme, the binding that takes place prior to execution, is disengaged immediately after execution in order to permit the 'system' to evolve for the next point of execution. Flexibility and personalization are achieved through a variety of service providers offering functionality through a competitive market-place, with each software provision being accompanied by explicit properties of concern for binding (e.g. dependability, performance, quality, license details etc).

A component is simply a reusable software executable. Our *serviceware* clearly includes the software itself, but in addition has many non-functional attributes, such as cost and payment, trust, brand allegiance, legal status and redress, security and so on. Binding requires us to negotiate across all such attributes (as far as possibly electronically) to establish a binding, at the extreme just before execution.

Requirements for software need to be represented in such a way that an appropriate service can be *discovered* on the network. The requirements must convey therefore both the description and intention of the desired service. Given the highly dynamic nature of software supplied as a service, the maintainability of the requirements representation becomes an important consideration. However, the aim of the architecture is not to prescribe such representation, but support whatever conventions users and service suppliers prefer.

Automated *negotiation* is another key issue for research, particularly in areas where non-numeric terms are used e.g. legal clauses. Such clauses do not lend themselves to offer/counter-offer and similar approaches. In relation to this, the structure and definition of profiles and terms needs much work, particularly where terms are related in some way (e.g. performance and cost). Also we need insight to the issue of when to select a service and when to enter negotiations for a service. It is in this area that multi-disciplinary research is planned. We plan to concentrate research in these areas, and use as far as possible available commercial products for the software infrastructure.

Finally, many issues need to be resolved concerning mutual performance monitoring and claims of legal redress should they arise.

## **10. Conclusions**

We have presented a new staged model of the software lifecycle, motivated by the need to formulate an abstraction that is supported partly by empirical published evidence, and partly by the authors' field experiences. In particular, we have observed how the "project knowledge" has been progressively lost over the lifecycle, and the enormous implications for our ability successfully to support the software. We have argued that by understanding the staged model, a manager can better plan and resource a

project, in particular to avoid it slipping into servicing irreversibly. We also indicated some future directions based on this approach.

### **Acknowledgements**

Keith Bennett would like to thank members of the Pennine Research Group at Durham, UMIST and Keele Universities for the collaborative research which has led to the author's input to this chapter (in particular Malcolm Munro, Paul Layzell, Linda Macauley, Nicolas Gold, Pearl Brereton, and David Budgen). He would also like to thank the Leverhulme Trust, BT and EPSRC for generous support, and Deborah Norman for help in preparing the chapter.

Vaclav Rajlich would like to thank Tony Mikulec from Ford Motor Co. for generous support of research in software maintenance. Also discussions with Franz Lehner while visiting University of Regensburg, and with Harry Sneed on several occasions influenced the author's thinking about this area.

Norman Wilde would like to thank the support over the last 15 years of the Software Engineering Research Center (SERC); its industrial partners have taught him most of what he knows about software maintenance. More recently the US Air Force Office of Scientific Research under grant F49620-99-1-0057 has provided an opportunity to study the FASTGEN system mentioned as one of the case studies.

## References

- [1] IEEE *Standard Glossary of software engineering terminology*, standard IEEE Std 610.12-1990, available from IEEE, Los Alamitos. **Also IEEE Software engineering - IEEE standards collection** New York: IEEE, Edition 1994 ISBN/ISSN 155937442X.
- [2] McDermid, J.A. (Ed.) *The Software Engineer's Reference Book*, Butterworth-Heinemann, 1991, ISBN 0-750-61040-9.
- [3] Royce, W.W. *Managing the Development of Large Software Systems*, Proc. IEEE WESCON 1970, IEEE, pp.1-9 (reprinted in Thayer, R.H. (Ed.) IEEE Tutorial on Software Engineering Project management).
- [4] Boehm, B.W. *A spiral model of software development and enhancement*, IEEE Computer, May 1988, pp.61-72.
- [5] Rajlich, V.T.; Bennett, K.H. *A staged model for the software lifecycle*, IEEE Computer, Vol. 33, No. 7, pp.66-71, July 2000, ISSN 0018-9162.
- [6] Pigoski, T.P. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley & Sons, Inc., New York 1997.
- [7] Lientz, B.; Swanson, E.B. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organisations*, Addison-Wesley, 1980.
- [8] Lientz, B.; Swanson, E.B.; Tompkins, G.E. *Characteristics of applications software maintenance*, Communications of the ACM, Vol. 21, pp.466-471.

- [9] Sommerville, I. *Software Engineering*, Addison Wesley; ISBN: 0201427656, 1995.
- [10] Pressman, R.S. *Software Engineering*, McGraw-Hill Publishing Company; ISBN: 0070521824, 1996.
- [11] Warren, I. *The Renaissance of Legacy systems*, Springer-Verlag UK; ISBN: 1852330600, 1999.
- [12] Foster, J.R. *Cost factors in software maintenance*, Ph.D. Thesis, Computer Science Department, University of Durham, 1993.
- [13] Robson, D.J.; Bennett, K.H.; Munro, M.; Cornelius, B.J. *Approaches to Program Comprehension*, Journal of Systems and Software, Vol. 14 (Feb. 1991) pp.79-84. Reprinted in "Software Re-engineering" (ed R Arnold. IEEE Computer Society Press, 1992.
- [14] IEEE *Standard for Software Maintenance*, IEEE, Los Alamitos, USA. ISBN 0-7381-0336-5, pp.56.
- [15] International Standards Organisation, *International Standard Information Technology: Software Maintenance*, (ISO/IEC 14764:1999) International Standards Organisation, 15 November 1999.
- [16] Wirth, N. *Program Development by Stepwise refinement*, Communications of the ACM, Vol.14, No. 4, April 1971.
- [17] Basili, V.R.; Turner, A.J. *Iterative Enhancement: A Practical Technique for Software Development*, IEEE Transactions on Software Engineering, SE-1, 4, pp.90-396 (1975). (An updated version was published as Auerbach Report 14-01-05, 1978, and in Tutorial on Software Maintenance published by the IEEE Computer Society (1982)).

- [18] Brooks, F. *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley Publishing Company; ISBN: 0201835959.
- [19] Lehman, M.M.; Belady, L.A. *A model of large program development*, IBM Syst. J. Vol.3, pp.225-252, 1976.
- [20] Burch, E.; Kunk, H. *Modeling software maintenance requests: a case study*, Proc. IEEE Int. Conf. On Software Maintenance 1997, IEEE Computer Society Press, pp.40-47.
- [21] Lehman, M.M., *Programs, lifecycles, and the laws of software evolution*, IEEE Trans. Software Engineering, Vol. 68, No. 9, pp.1060-1076, 1980.
- [22] Lehman, M.M. *Program evolution*, Information Processing Management, Vol. 20, pp.19-36, 1984.
- [23] Lehman, M.M. *Program evolution*, Academic Press, London. 1985.
- [24] Lehman, M.M. *Uncertainty in Computer Application and its Control through the Engineering of Software*, Journal of Software Maintenance, Vol. 1, No. 1, pp.3-28, Sept.1989.
- [25] Lehman, M.M.; Ramil, J.F. *Feedback, Evolution And Software Technology - Some Results from the FEAST Project*, Keynote Lecture, Proc. 11<sup>th</sup> Int. Conf. on Software Engineering and its Application, Vol. 1, Paris, 8-10 Dec. 1998, pp.1-12.
- [26] Ramil, J.F.; Lehman, M.M.; Kahen, G. *The FEAST Approach to Quantitative Process Modelling of Software Evolution Processes*, Proc. PROFES'2000 2<sup>nd</sup> International Conference on Product Focused Software Process Improvement, Oulu, Finland, 20-22 Jun. 2000, in Frank Bomarius

- and Markku Oivo (eds.) LNCS 1840, Springer Verlag, Berlin, 2000, pp.311-325. This paper is a revised version of the report: Kahen, G.; Lehman, M.M.; Ramil, J.F. Model-based Assessment of Software Evolution Processes, Research Report 2000/4, Dept. of Comp., Imp. Col., Feb. 2000.
- [27] Lehman, M.M.; Perry, D.E.; Ramil, J.F. ***Implications of Evolution Metrics on Software Maintenance***, Int. Conf. on Soft. Maintenance (ICSM'98), Bethesda, Maryland, Nov. 16-24, 1998, pp.208-217.
- [28] Ramil, J.F.; Lehman, M.M. ***Challenges facing Data Collection for Support and Study of Software Evolution Processes***, position paper, ICSE 99 Workshop on Empirical Studies of Software Development and Evolution, Los Angeles, May 18, 1999, 5 pps.
- [29] Sneed, H.M., translator Johnson, I. ***Software Engineering Management***, Ellis Hoewood LTD., Chichester, West Sussex, 1989, pp.20-21, original German edition Software Management, Rudolf Mueller Verlag, Koln, 1987.
- [30] Lehner, F. ***The software lifecycle in computer applications***, Long Range Planning, Vol. 22, No. 5, (Pergamon Press) 1989, pp.38-50.
- [31] Lehner, F. ***Software lifecycle management based on a phase distinction method***, Microprocessing and Microprogramming, Vol. 32 (North Holland), pp.603-608, 1991.
- [32] Truex, D.P.; Baskerville, R.; Klein, H. ***Growing Systems in Emergent Organizations***, C.A.C.M. Vol. 2, No. 8, Aug. 1999.

- [33] Cusumano, M.; Yoffe, D. **Competing on Internet Time – Lessons from Netscape and its Battle with Microsoft**, Free Press (Simon & Schuster) 1998.
- [34] Bennett, K.H. **Legacy Systems: Coping with success**, IEEE Software Vol. 12, No. 1, pp.19-23, Jan. 1995.
- [35] Henderson, P. (Ed.) **Systems Engineering for Business Process Change**, Springer, 2000, ISBN 1-85233-222-0.
- [36] **Systems Engineering for Business process Change**, UK EPSRC (May 1999). URL at <http://www.staff.ecs.soton.ac.uk/~ph/sebpc>
- [37] Pfleeger, S.L.; Menezes W. **Technology transfer: marketing technology to software practitioners**, (Submitted to IEEE Software) 1999.
- [38] Naur, P.; Randell, B. (eds.) **Software Engineering Concepts and Techniques**, NATO Science Committee, Proc. NATO Conferences, Oct. 7-11, 1968, Garmisch, Germany. Petrocelli/Charter (New York).
- [39] Shaw, M.; Garland, D. **Software Architectures**, Prentice Hall. ISBN 0131829572, 1996.
- [40] Beck, Kent, **Embracing Change with Extreme Programming**, IEEE Computer, Vol. 32, No. 10, pp.70-77, October 1999.
- [41] **The architecture of Open VME**, ICL publication ref. 55480001, from ICL, Cavendish Rd., Stevenage, Herts, UK SG1 2DY, 1994.
- [42] Cusumano, M.A.; Selby, R.W. **Microsoft Secrets**, HarperCollins, 1997, ISBN: 0006387780.

- [43] Jacobson, I.; Booch, G.; Rumbaugh, J. *The Unified Software Development Process*, Addison Wesley, 1999.
- [44] Booch, G. *Developing the Future*, Communications of the ACM, Vol. 44, No. 3, March 2001, pp.119-121.
- [45] Parnas, D.L. *Software Aging*, Proc. 16<sup>th</sup> Int. Conf. On Software Engineering, IEEE Computer Society Press, 1994, pp.279-287.
- [46] Eick, S.G.; Graves, T.L.; Karr, A.F.; Marron, J.S.; Mockus, A. *Does Code Decay? Assessing Evidence from Change Management Data*, IEEE Trans. On Software Engineering, Jan. 2001, pp.1-12.
- [47] Rajlich, V.; Wilde, N.; Buckellew, M.; Page, H. *Software Cultures and Evolution*, to be published in IEEE Computer, Sept. 2001.
- [48] Johnson, J.H. *Substring Matching for Clone Detection and Change Tracking*, Proc. IEEE International Conference on Software Maintenance, Victoria, Canada, September 1994, pp. 120-126.
- [49] Baxter, D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. *Clone Detection Using Abstract Trees*, IEEE International Conference on Software Maintenance, 1998, pp. 368-377.
- [50] Lagu, B.; Proulx, D.; Mayrand, J.; Merlo, E.M.; Hudepohl, J. *Assessing the Benefits of Incorporating Function Clone Detection in a Development Process*, IEEE International Conference on Software Maintenance, 1997, pp. 314-231.
- [51] Burd, E.; Munro, M. *Investigating the Maintenance Implications of the Replication of Code*, IEEE International Conference on Software Maintenance, 1997, pp. 322-329.

- [52] Olsem, M.R. *An Incremental Approach to Software Systems Re-engineering*, Software Maintenance: Research and Practice 10, 1998, pp.181-202.
- [53] Canfora, G.; De Lucia, A.; Di Lucca, G. *An Incremental Object-Oriented migration Strategy for RPG Legacy Systems*, International Journal of Software Engineering and Knowledge Engineering, Feb. 1999, Vol. 9, No. 1, pp.5-25.
- [54] Tamai, T.; Torimitsu, Y. *Software lifetime and its Evolution Process over Generations*, Proc. IEEE International Conference on Software Maintenance 1992, pp.63-69.
- [55] Kappelman, L.A. *Some Strategic Y2K blessings*, IEEE Software, pp.42-46, April 2000.
- [56] Fanta, R.; Rajlich, V. *Reengineering Object-Oriented Code*, Proc. IEEE International Conference on Software Maintenance, 1998, pp.238-246.
- [57] Fanta, R.; Rajlich, V. *Removing Clones from the Code, to be published in Journal of Software Maintenance*.
- [58] Wilde, N.; Buckellew, M.; Page, H.; Rajlich, V. *A Case Study of Feature Location in Unstructured Legacy Fortran Code*, Proceedings CSMR'01, IEEE Computer Society, March 2001, pp.68-76.
- [59] Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison Wesley, Reading, Massachusetts, 1999.
- [60] Yau, S.S.; Collofello, J.S.; MacGregor, T., *Ripple Effect Analysis of Software Maintenance*, Proc. IEEE COMPSAC, 1978, pp.60-65.

- [61] Rajlich, V. ***Modeling Software Evolution by Evolving Interoperation Graphs***, Annals of Software Engineering, Vol. 9, 2000, pp.235-248.
- [62] Ogando, R.M.; Yau, S.S.; Liu, S.S.; Wilde, N. ***An Object Finder for Program Structure Understanding in Software Maintenance***, Journal of Software Maintenance: Research and Practice, Vol. 6, pp.261-283, 1994.
- [63] Chapin, N.; Lau, T. S. ***Effective Size: An Example of USE from Legacy Systems***, Journal of Software Maintenance: Research and Practice, Vol. 8, pp.101-116, 1996.
- [64] Oman, P. ***Maintenance Tools***, IEEE Software, pp.59-65, May 1990.
- [65] Von Mayrhauser, A.; Vans, A.M. ***Program Comprehension During Software Maintenance and Evolution***, IEEE Computer, pp.44-55, August 1995.
- [66] Littman, D.C.; Pinto, J.; Letovsky, S.; Soloway, E. ***Mental Models and Software Maintenance***, Empirical Studies of Programmers, Ed. Soloway, E.; Iyengar, S. pp.80-98, Norwood, N.J.: Ablex, 1986.
- [67] Basili, V.R.; Mills, H.D. ***Understanding and Documenting Programs***, IEEE Transactions on Software Engineering. March 1982. Vol. SE-8, No. 3, pp.270-283.
- [68] Younger, E.J.; Bennett, K.H. ***Model-Based Tools to Record Program Understanding***, Proceedings of the IEEE 2<sup>nd</sup> International Workshop on Program Comprehension, July 8-9, 1993, Capri, Italy, IEEE Computer Society Press. pp.87-95.
- [69] Standish, T.A. ***An Essay on Software Reuse***, IEEE Transactions on Software Engineering, September 1984, Vol. SE-10, No. 5, pp.494-497.

- [70] Weiser, M.; Lyle, J. *Experiments on Slicing-Based Debugging Aids*, Empirical Studies of Programmers, Albex, Norwood NJ, 1986, pp.187-197.
- [71] Storey, M.A.D.; Fracchia, F.D.; Müller, H.A. *Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization*, Proceedings of the 5<sup>th</sup> IEEE International Workshop on Program Comprehension, pp.17-28, May 28-30, 1997.
- [72] Brooks, R. *Toward a Theory of Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, No. 6, pp.542-554, 1983.
- [73] Soloway, E.; Ehrlich, K. *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. SE10, No. 5, pp.595-609, September 1984.
- [74] Pennington, N. *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Cognitive Psychology, Vol. 19, No. 3, pp.295-341, July 1987.
- [75] Letovsky, S. *Cognitive Processes in Program Comprehension*, Journal of Systems and Software, Vol. 7, pp.325-339, 1987.
- [76] Von Mayrhauser, A.; Vans, A.M.; Howe, A.E. *Program Understanding Behaviour during Enhancement of Large-scale Software*, Journal of Software Maintenance: Research and Practice, Vol. 9, pp.299-327, 1997.
- [77] Shneiderman, B.; Mayer, R. *Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results*, International

Journal of Computer and Information Sciences, 1979, Vol. 8, No. 3, pp.219-238.

[78] Knight, C.; Munro, M. *Comprehension with[in] virtual environment visualisations*, Proc. IEEE 7<sup>th</sup> Int. Workshop on Program Comprehension, May 5-7, 1999, pp.4-11.

[79] Biggerstaff, T.; Mitbander, B.; Webster, D. *Program Understanding and Concept Assignment Problem*, Communications of ACM Vol. 37, No. 5, pp.72-83 (May 1994).

[80] Wilde, N.; Scully, M. *Software Reconnaissance: Mapping Program Features to Code*, Journal of Software Maintenance: Research and Practice, Vol. 7, (1995) pp.49-62.

[81] Chen, K.; Rajlich, V. *Case Study of Feature Location Using Dependency Graph*, Proc. International Workshop on Program Comprehension, IEEE Computer Society Press, 2000, pp.241-249.

[82] Hager, J.A. *Developing Maintainable Systems: A Full Life-Cycle Approach*, Proc. Conference on Software Maintenance - 1989, IEEE Computer Society Press, October 16-19, 1989, pp.271-278.

[83] Parnas, D.L. *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 1972, Vol. 29, pp.1053-1058.

[84] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.

[85] Wilde, N.; Casey, C. *Early Field Experience with the Software Reconnaissance Technique for Program Comprehension*, Proceedings

International Conference on Software Maintenance - ICSM'96, IEEE Computer Society, November 1996, pp.312-318.

[86] Wilde, N.; Huitt, R. *Maintenance Support for Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No. 12, (December 1992), pp.1038-1044.

[87] Wilde, N.; Knudson, D. *Understanding Embedded Software Through Instrumentation: Preliminary Results from a Survey of Techniques*, Report SERC-TR-85-F, Software Engineering Research Center, Purdue University, 1398 Dept. of Computer Science, West Lafayette, IN 47906, February, 1999. Available at [http://www.cs.uwf.edu/~wilde/publications/TecRpt85F\\_ExSum.html](http://www.cs.uwf.edu/~wilde/publications/TecRpt85F_ExSum.html).

[88] Wilde, N.; Casey, C.; Vandeville, J.; Trio, G.; Hotz, D. *Reverse Engineering of Software Threads: A Design Recovery Technique for Large Multi-Process Systems*, Journal of Systems and Software, 1998, Vol. 43, pp.11-17.

[89] Wendorff, P. *Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project*, Proceedings Fifth European Conference on Software Maintenance and Reengineering - CSMR'01, IEEE Computer Society Press, March 2001, pp.77-84

[90] Pigoski, T.M.; Sexton, J. *Software Transition: A Case Study*, Proc. Conference on Software Maintenance - 1990, IEEE Computer Society Press, November 1990, pp.200-204.

- [91] Vollman, T. *Transitioning From Development to Maintenance*, Proc. Conference on Software Maintenance - 1990, IEEE Computer Society Press, November 1990, pp.189-199.
- [92] Pigoski, T.M.; Cowden, C.A. *Software Transition: Experience and Lessons Learned*, Proc. Conference on Software Maintenance - 1992, IEEE Computer Society Press, November 1992, pp.294-298.
- [93] Rajlich, V. *Incremental Redocumentation Using the Web*, IEEE Software, September/October 2000, pp.102-106.
- [94] Bollig, S.; Xiao, D. *Throwing Off the Shackles of a Legacy System*, IEEE Computer, 1988, Vol. 31, No. 6, June 1988, pp.104-109.
- [95] Bennett, K.H.; Layzell, P.J.; Budgen, D.; Brereton, O.P.; Macaulay, L.; Munro, M. *Service-Based Software: The Future for Flexible Software*, IEEE APSEC2000, The Asia-Pacific Software Engineering Conference, 5-8 December 2000, Singapore, IEEE Computer Society Press, 2000.
- [96] Bennett, K.H.; Munro, M.; Brereton, O.P.; Budgen, D.; Layzell, P.J.; Macaulay, L.; Griffiths, D.G.; Stannet, C. *The future of software*, Communications of ACM, Vol.42, No. 12, pp.78-84. (Dec. 1999).
- [97] Bennett, K.H.; Munro, M.; Gold, N.E.; Layzell, P.J.; Budgen, D.; Brereton, O.P. *An Architectural Model for Service-Based Software with Ultra Rapid Evolution*, Proc. IEEE Int. Conf. On Software Maintenance, Florence, 2001 (to appear).
- [98] Lovelock, C.; Vandermerwe, S.; Lewis, B. *Services Marketing*, Prentice Hall Europe, 1996, ISBN 013095991X.