Software Maintenance and Evolution: a Roadmap

Conference Paper · May 2000

DOI: 10.1145/336512.336534 · Source: DBLP

2 authors, including:

Vaclav Rajlich
Wayne State University
102 PUBLICATIONS **2,615** CITATIONS

SEE PROFILE

Software Maintenance and Evolution

K. H. Bennett
Research Institute for Software Evolution
University of Durham
UK
DH1 3LE
keith.bennett@durham.ac.uk

V. T. Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202
rajlich@cs.wayne.edu

Aims and Objectives

Software maintenance and evolution are characterised by their huge cost and slow speed of implementation. Yet they are inevitable activities – almost all software which is useful and successful stimulates user-generated requests for change and improvements. Our aim is to describe a landscape for research in software maintenance and evolution over the next ten years, in order to improve the speed and accuracy of change while reducing costs, by identifying key problems, promising solution strategies and topics of importance. The aim is met by taking two approaches. Firstly current trends and practices are projected forward using a new model of software evolution called the *staged model*. Both strategic problems and research to solve particular tactical problems are described within this framework. Secondly, a longer term, and much more radical vision of software evolution is presented. Both general principles and specific research topics are provided, both within an overall strategy of engineering research and rationale.

1 State of the art and industrial context in maintenance and evolution

1.1 Basic definitions

Software maintenance is defined in IEEE Standard 1219 [IEEE93] as:

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

A similar definition is given by ISO/IEC [ISO95], again stressing the post-delivery nature:

The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity.

The term *software evolution* lacks a standard definition, but some researchers and practitioners use it as a preferable substitute for maintenance. In this chapter, we shall use *maintenance* to refer to general post-delivery activities, and *evolution* to refer to a particular phase in the staged model described in Section 2.

Pioneering work (that is still relevant to basic understanding) was undertaken by Lehman, who carried out empirical experiments on OS360 using a sequence of releases [LEH80]. This has set a good precedent for the field: very small programs do not have maintenance problems, and research results *must* scale up to industrial applications for them to be useful [McDER99], so that research in maintenance needs to be undertaken in collaboration with industry. Some of the practical consequences of approaches which work in the realities of an industrial context are reported by Sneed [..]. Research in software maintenance has been undertaken in seven broad areas:

1. System dynamics, to model the software as it changes over time, in order better to understand the underlying mechanisms.
2. Maintenance processes; defining, measuring, improving, risk analysis, quality assurance.
3. Studies of software change; impact analysis, change propagation.
4. Products, linking software attributes to maintainability (from architecture to identifier naming conventions); higher levels of software abstraction.
5. Program comprehension methods and tools, to link attributes to better cognitive understanding.
6. High level management, business and people issues; business models such as outsourcing and applications management.
7. Legacy and reverse engineering, to recover a software asset that has become very hard (expensive) to maintain.
8. Validation, ensuring that the software changes work as required, and the unchanged parts have not become less dependable.

In this chapter, two approaches to developing a ‘road-map’ of research and of developments in the field are used. The first approach projects and extrapolates from current trends and problems. The map is facilitated by a novel *staged model* of software development that is based on empirical observation. This helps an analysis of maintenance and evolution for the modern environment for software development that stresses components, the internet and distributed systems.

The second approach represents the outcome of a process that has involved experts from a range of disciplines brainstorming the problem, over a period of years. Thinking ‘outside the box’ was positively welcomed. The intention has been that a radical improvement in software maintenance will need a radical new way of thinking about it. Both the process and its outcome are described in detail in [BRER99].

Other very different approaches may of course increase in importance. One such approach is the ‘open source’ movement, initiated in the UNIX world. Maintenance as

a collaborative, cultural activity has shown major benefits, certainly in terms of software reliability (e.g. LINUX) and performance.

Revalidation and testing are required after changes are made to software. These activities can be very time consuming and are an integral part of maintenance; they are considered further elsewhere in this book.

The term *maintenance* is also currently being applied to the problem of keeping web pages up to date and consistent. There will surely be a range of such problems, to which generic research solutions can be applied in addition to domain specific issues.

1.2 Importance of maintenance

A very widely cited survey study by Lientz and Swanson in the late 1970s, and repeated by others in different domains, exposed the very high fraction of life-cycle costs that were being expended on maintenance. Lientz and Swanson categorised maintenance activities into four classes:

- Adaptive – changes in the software environment
- Perfective – new user requirements
- Corrective – fixing errors
- Preventive – prevent problems in the future.

Of these, the survey showed that around 75% of the maintenance effort was on the first two types, and error correction consumed about 21%. Many subsequent studies suggest a similar magnitude of the problem. These studies show that the incorporation of *new user requirements* is the core problem for software evolution and maintenance.

If changes can be anticipated at design time, they can be built in by some form of parameterisation. The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even *conceive* of. So software maintenance is important because (i) it consumes a large part of the overall lifecycle costs (ii) the inability to change software quickly and reliably means that business opportunities are lost. These are *enduring problems*, so that the profile of maintenance research is likely to increase over the next ten years.

1.3 What is software maintenance?

Despite the large expenditure, little is known about the empirical nature of software maintenance, in terms of its effect on the artefact, on the process and on the software engineers and users. The first vista in the research landscape is therefore:

- To gain more empirical information about the nature of software maintenance, in terms of its effect on the software itself, on processes, on organisations and people. What actually happens from release to release? For example, in Cusumano and Selby it was reported that a feature set during each iteration may change by 30% or more, as a direct result of the team learning process during the iteration [CUSU97]. Lehner [LEHN91] described yearly variations in the frequency of the changes of a long lived system. Is it possible to describe the changes in terms of a set of basic operations? More empirical work is crucial to inform progress on better maintenance processes.

- To express such understanding in terms of predictive models which can be validated through experiment. The models may inform both the technical and business facets of maintenance (e.g. risk models). For example, Lehman is using feedback control models in the FEAST project [LEHM98].
- To explore and formalise the relationships between technology and business models (for example, the implications of outsourcing and applications management, or the technical and business views of legacy software management).
- To understand how such models may be exploited in an industrial context (for example, in cost estimation). This work should lead to better metrics.
- To establish accepted evaluation procedures for assessing new developments and processes, in terms of the implications for maintenance, especially in an industrial context on large scale applications.

The final point can be generalised: often new technologies are proposed and introduced without consideration of what happens when the software has to be changed. If such innovations are to be exploited successfully, the full lifecycle needs to be addressed, not just the initial development. For example, object oriented technology was considered to be 'the solution to software maintenance'; empirical evidence is now showing that OO is creating its own new maintenance problems, and has to be used with care (e.g. by keeping inheritance under control) to ensure that maintenance is not even more difficult than for traditional systems. Recent technologies such as agents, components, graphical user interfaces, and modern ideas of logical, constraint, real-time and concurrent programming and so on need to be explored from a maintenance perspective.

Better understanding via such models should help researchers devise a much better definition of *maintainability*; currently this is a very poorly defined term of very limited use in industry.

A major challenge for the research community is to develop a good theoretical understanding and underpinning for maintenance and evolution, which scales to industrial applications. Most computer science research has been of benefit to the initial development of software. Type theory and configuration management have in different ways made major contributions to maintenance. Many others claim to do so, but reliable empirical evidence is lacking.

1.4 Structure of the chapter

Section 2 of the chapter explains the staged model of the software lifecycle. Section 3 explores software change in more detail. Section 4 deals with the problems of legacy systems. Section 5 deals with emergent organisations and the challenges they represent. Section 6 contains the conclusions.

2 Research framework: a staged model for software lifecycle

2.1 Introduction

The conventional analysis (of Lientz and Swanson) is no longer useful for modern software development. It does not help with reasoning about component-based

systems, distributed systems etc. and does not help with planning software evolution. In [BENN99], it is argued, based on empirical observation, that the activities undertaken during software evolution vary greatly. This is in contrast to the standard definitions offered in section 1, where maintenance was seen as a single post-delivery activity. A novel staged model is therefore now introduced, comprising five distinct stages. The model is summarised below and is seen as an essential framework in which to identify research needs and areas.

2.2 Model outline

The staged model of software lifecycle was introduced in [BENN99] and is summarized in Figure 1. It represents the software lifecycle as a sequence of stages, with *initial development* being the first stage. Its key contribution is to separate the 'maintenance' phase into an *evolution* stage followed by a *servicing* and *phase out* stages.

During the initial development, the *first version* of the software system is developed. That first version may be lacking some features, but it already possesses the *architecture* that will persist thought the rest of the life of the program. In one documented instance, we studied a program that underwent substantial changes during its 20 years of existence [HOLT94], but it still possesses the architecture of the original first version.

Another important outcome of the initial development is the *knowledge* that the programming team acquires: the knowledge of the application domain, user requirements, role of the application in the business process, solutions and algorithms, data formats, strengths and weaknesses of the program architecture, operating environment, etc. This knowledge is a crucial prerequisite for the subsequent phase of *evolution*.

Software *evolution* takes place only when the initial development was successful. The goal is to adapt the application to the ever-changing user requirements and operating environment. The evolution stage also corrects the faults in the application and responds to both developer and user learning, where more accurate requirements are based on the past experience with the application. The inevitability of evolution is documented in [LEHM85]. In business terms, the software is being evolved because it is successful in the marketplace; revenue streams are buoyant, user demand is strong, the development atmosphere is vibrant and positive, and the organization is supportive. Return on investment is excellent.

Both software architecture and software team knowledge make evolution possible. They allow the team to make substantial changes in the software without damaging the architectural integrity. Once one or the other aspect disappears, the program is no longer evolvable and enters the stage of *servicing* (also called software maturity [LEHN91]). During the servicing stage, only small tactical changes (patches, code changes and wrappers) are possible.

For the business, the software is likely to be no longer a core product, and the cost-benefit of changes are much more marginal.

There is a positive feedback between the loss of software architecture coherence, and the loss of the software knowledge. Less coherent architecture requires more extensive knowledge in order to evolve it. However if the knowledge necessary for evolution is lost, the changes in the software will lead to a faster deterioration of the architecture. Very often on software projects, the loss of knowledge is triggered by loss of key personnel, and the project slips into the servicing stage. We call this process *code decay*.

The reversal from servicing stage back to evolution stage is a worthy research goal, but at this point we are unaware of any real-life project that have successfully accomplished that. It is not simply a *technical* problem; the *knowledge* of the software team must also be addressed. For all practical reasons, the transition from evolution to servicing is irreversible.

As mentioned above, during the servicing stage only minor tactical program changes can be done. They usually take a form of patches and wrappers, and they further deteriorate the architecture.

The final stages are *phase-out* and *close-down*. During phase-out, no more servicing is being undertaken, but the system still may be in production. The users must work around known deficiencies. During close-down the software use is disconnected and the users are directed towards a replacement.

An amplification of the staged model is the *versioned staged model* of Figure 2 [Benn99]. In it, the software team produces versions of the software during an extended phase of evolution, but the versions are no longer evolved, only serviced. All substantial changes in the functionality are implemented in the future versions. If a version becomes outdated and the users need a new functionality, they have to replace their version with a new one. The so-called "shrink-wrap software" sold by software companies to large user communities often follows this versioned staged model.

We conclude from a research perspective:

- Each stage has very different technical solutions, processes, staff needs and management activities. We now have the opportunity to *explore and research the best solution for each stage*. It is very clear (for example) that solutions for evolution and for servicing are radically different.
- A key issue is the nature of the *stage changes and boundaries*, and better understanding of these and their characteristics and information flow across them will enable managers to plan better.
- Better understanding of how to keep a system *within a particular stage* for as long as possible is of practical importance.

2.3 Initial development

The key research challenge is to find ways of developing software such that the software can be more easily and reliably changed in subsequent phases (of course, in a typical project, the architecture changes through initial development as engineering alternatives are explored and rejected; but well before the end of initial development the architecture should be rigorously defined). This grand research challenge is often

expressed as ‘the cost of making the change is proportional to the size of the change, not to the size of the overall software system’. It necessarily includes consequential re-verifying and re-validating the software as well as implementing the changes. As already noted, if changes can be predicted, they can be anticipated in the design. The hard problem is coping with unanticipated changes.

The two key outcomes of initial development are (i) the architecture and (ii) the team’s knowledge. An area of considerable current research activity is *architecture definition languages* (ADLs), to make explicit in a formal way the architecture. Much less research has been done on addressing the skills of software architects themselves. It is known that they require skills different to those of programmers, but it is also clear that a good architect is very highly talented, much knowledge is *tacit*, and attempts to automate this via knowledge based systems should proceed with caution. However, good practice in architecture can be formalized, and work on *patterns* (i.e. representations of standard ways of doing things) looks very promising. This should be of particular help when the architecture is not innovative, but is very similar to previous cases.

The staged model suggests that the central aim of flexible software is to assist the next (evolution) stage, not subsequent stages. So ‘design for change’ is predominantly aimed at strategic evolution, not code level servicing; the research considerations are at the same level of abstraction in evolution and in initial development.

It is generally considered that software architecture research will play a major part in achieving this. Formalisation of architectures will provide explicit representations. However, it is very interesting to note that no current definition of software architecture includes the time dimension (compare, for example, with the ideas in architecture of buildings in [BRAN94]). At the other end of the scale, benefit will be obtained from using technology which is ‘neutral’, in the sense that the technology can be changed easily without consequential affects on the software design. For example, interpreted intermediate languages (as for Java) allow platform independence; we may expect this trend to continue.

2.4 Evolution stage

The aim is to implement (and revalidate) possibly major changes to the system without being able *a priori* to predict how user requirements will evolve. In terms of project management, initial development and evolution are clearly closely linked, and the key issue is to keep the senior architects and designers in the team. If they are lost, then the software very quickly and irreversibly moves to the servicing stage. Outsourcing at this stage is very difficult, unless the team also moves. The project is *core* to the organisation in terms of business strategy and profit. It seems that understanding how to cope with major changes in user requirements and yet minimise the integrity of the system architecture is a task at which humans are expert and which is difficult to automate. There is a vision of being able to add extra capabilities – both functional and non-functional – to a software system, that do not corrupt system invariants but add to or enhance them. This also leads to the *feature interaction* problem, which is a fruitful area for research. Equally, there is no great merit in proposing system understanding tools and methods; such understanding is present in the existing team.

Three research topics are identified:

1. Architectures which will allow considerable unanticipated change in the software without compromising system integrity and invariants.
2. Architectures which themselves can evolve in controlled ways.
3. Managing the knowledge and expertise of the software team.

A highly promising area of research is therefore to find ways to raise the level of abstraction in which evolution is expressed, reasoned about and implemented. Even where very high level abstractions were used during design, maintenance today is still performed in reality using source code. Research topics therefore include:

- raising the abstraction level of the language used.
- separating declarative issues like business objects from control issues.
- representing domain concepts in domain analysis and domain specific languages.
- partitioning architectures into independently evolving subsystems

Business research on the evolution stage is also needed. For example, product evolution must start during the success of the old product**.** This is not a technical imperative but an economic one; the management and business case has been made by researchers such as Handy [HAND94], and is not restricted to the software industry (Handy provides a number of non-software examples). This is the fundamental reason why software undergoes releases and new versions. Deeper understanding of the business imperatives for modern software based industry is needed. The sophistication and understanding of modern product, marketing and service approaches used in many other industries is largely absent in the software industry.

2.5 Servicing stage

The aim is to implement and test tactical changes to the software, undertaking this at minimum cost and within the capabilities of the staff available. The move to servicing involves a big change to project management:

- Only minor corrections, enhancements and preventative work should be undertaken
- Senior designers and architects do not need (and are unlikely) to be available
- The staff do not require the same level of domain engineering or software engineering expertise
- Tools and processes are very different
- A typical engineer will be assigned only part of the software to support, and thus will have partial knowledge of the system.
- The process is (or should be) now stable, well understood and mature. Its iterative nature means it is well suited to process improvement, and measurement.
- Accurate cost prediction is needed.

The processes of servicing are well understood. The key problems become:

- Mapping the change as expressed by the user (usually in terms of behaviour) to the software.
- Understanding enough of the software to make the change accurately.

- Making the change without introducing unexpected additional effects.
- Revalidating the change.

Program comprehension is the central research problem and is amenable to much better tool support; once the change and its impact have been understood, it is relatively simple to make it. Tool research is relevant to:

- Impact analysis and ripple effect management.
- Display of program structure (call graph, dominance tree, class structure etc).
- Regression testing.
- Better programming language design.
- Concept identification, location and representation.
- Configuration management and version control for large distributed systems.

In a large distributed system, the determination of which source modules or components form part of the system is an interesting problem; the components may be shared with other systems. Mendoca [1999] has raised the issue of the comprehension of the 'run time architecture' in a multi-thread concurrent system with dynamically created processes. In a large distributed system, it is not sensible to try to maintain the program in a conventional way i.e. halt it, edit the source, and re-execute it. It will be vital to be able to replace components on the fly, as stopping a large distributed system with many threads is not an option. A promising area for research is the visualization of software using non-software metaphors to aid cognitive understanding [KNIG99].

Research *on automated tool support* to *improve* the code (and test suites) in order to reduce the costs of servicing is needed. Fruitful work is anticipated in:

- Migration from obsolete to modern programming languages.
- Migration from obsolete to modern data bases.
- Restructuring code and data to remove unnecessary complexity (particularly complexity which has been introduced by heavy servicing).
- Metrics and evaluation methods to assess empirical issues.
- Documentation tools to manage comments (the Java documentation tools are a simple example).
- Delivery of service packs for shrink wrapped and consumer software.
- Upgrading software without the need to halt it.
- Programming language health checkers for susceptible constructs.
- Name and Identifier management.

Associated *economic models* will help to justify cost-benefit arguments for purchasing, and using such tools.

A large problem exists with software that is constructed for a mass market. It has already been noted that a company such as Microsoft cannot sensibly manage issuing small increments to 'shrink-wrapped' software; there is no means of ensuring that all users receive or take advantage of the update (which Microsoft call 'service packs'). Yet issuing a complete new version is unsatisfactory in this market. Similar problems are expected with consumer goods such as software-upgradeable mobile telephones. A sequence of upgrades (not all purchased by the consumer) may have to be compatible and inter-work.

A further problem is posed by servicing components. Software components are not immune from defects, and it is already the case that new versions, which may or may not be compatible with previous versions have to be introduced, with the possibility of introducing new defects. Industry has to spend resource on 'component harnesses' which explore the actual behaviour of components (and the source code may not be available). Research is needed on the best way to manage this.

It is possible to outsource the servicing. An advantage of the staged model is that it clarifies the relationship between software user and the vendor/service company. Research is needed in service level agreements that have clear, 'no-surprise' effects for both the customer and service company.

2.6 Phase-out and close down stages

The aim is to manage the software towards the end of its life. This would not seem a promising area for useful research, but there are two important business issues:

- If the software is outsourced, and the contract is nearing its end, how should the asset value of the software be managed?
- Can any of the software (and software team) be re-used?

3 Software change

Software change is the basic operation of both software evolution and software servicing. The two stages are separated by the difficulty of the change, allowing substantial changes during the evolution and only limited changes during the servicing; nevertheless the repeated change is the basic building block from which both evolution and servicing derive. The larger issues of evolvable architectures, code decay, etc. profit from the more detailed study of the properties of the individual change. So this activity will benefit from research, which will in turn benefit the wider issues raised earlier. The processes and methods that are now described are all aspects which will benefit from further research which in turn will lead to very substantial industrial benefit.

The change is a process that either introduces new requirements into an existing system, or modifies the system if the requirements were not correctly implemented, or moves the system into a new operating environment. It is called the *change minicycle* [ ] and consists of the following phases:

- Request for change
- Planning phase
    - Program comprehension
    - Change impact analysis
- Change implementation
    - Restructuring for change
    - Change propagation
- Verification and validation

- Redocumentation

A more precise definition of the minicycle process is still a subject of research.

A request for change often originates from the users of the system, and may have a form of a ‘bug report’ or a request for additional functionality. It is usually expressed in terms of the application domain concepts, for example: "Add a new feature to the student registration system so that students with *hold* on their record are not allowed to register".

Program comprehension is a prerequisite of the change and it has been a subject of extensive research [ ]. It has been reported that this phase consumes more than half of all maintenance resources [ ]. The program comprehension phase may be more important in the servicing stage because the knowledge of the team is more tactical and localised, and hence there is a greater need to invest in program comprehension before the change is implemented. A substantial part of program comprehension is location of the application domain concepts in the code, for example to find where in the code "course registration" is implemented.

Change impact analysis is the activity by which the programmers assess the extent of the change, i.e. the components that will be impacted by the change. Change impact analysis indicates how costly the change is going to be and whether it is to be undertaken at all.

After the preliminary phases establish feasibility of a change, the change is implemented. The change implementation may consist of several steps, each visiting one specific software component. If the visited component is modified, it may no longer fit with the other components because it may no longer properly interact with them. In that case secondary changes must be made in neighbouring components, which may trigger additional changes, etc. This process is called change propagation. Although each successful change starts and ends with consistent software, during the change propagation the software is often inconsistent.

If the current architecture does not support contemplated change, because the concepts of the application domain relevant to the change are delocalized in the code, the software should be restructured first. For example, if the concept "course registration" is widely delocalized in different components, than the change will be difficult because it will involve visits to all those components. The solution is to restructure first and to localize the concept in one location, and then to change it. The process employed is called *behaviour preserving transformation*. It does not change the behaviour of the program, but changes the architecture. In the case of difficult changes, an advisable strategy is to divide the change into two steps: firstly to transform the architecture so that the change will be localized; and then to make the change itself. This division of change into two steps makes the change easier than it would otherwise be.

The change minicycle ends with the update of the program documentation. However it is possible to do more at this point than just documentation update. If the documentation of the program is missing or incomplete, the end of the minicycle is the time to record the comprehension acquired during the change. Since program

comprehension consumes more than 50% of resources of software maintenance and evolution, it is a very valuable commodity. Yet in current practice, that value is thrown away when the programmer completes the change and turns his/her attention to new things. In order to avoid that loss, incremental and opportunistic redocumentation effort is called for. After a time, substantial documentation can be accumulated. [ ]

The phases of the change minicycle can be supported by specialized software tools. These tools are based on program analysis, which extracts important and relevant facts from the existing program, like call graph, data flows, etc. The extracted information is usually stored in a database and then used in the tools. The analysis can be in specific instances difficult because an accurate answer may be unsolvable and has to be approximated by heuristics. An example of such a difficult problem is pointer aliasing that may be needed in dataflow analysis and other contexts. The algorithms in that case are subject to research and the issues are accuracy and efficiency of the solution.

4 Software legacy and migration

Legacy software has been defined pragmatically as ‘software which is vital to our organization, but we don’t know what to do with it’ [BENN95]. Some years ago, this was one of the major software problems, but it has become less prominent recently. This is possibly because many organizations have been obliged to replace old software to ensure Y2K (millennium) compliance. However, it is safe to predict that the problem will soon appear again, in a much more severe form. Previously, legacy systems comprised mainly monolithic software, albeit in independent subsystems. Current software, based around distributed components from multiple vendors with ‘middleware’ support, and possibly within an enterprise framework is likely to be *far* more difficult to address. A much better understanding of the legacy problem has been acquired over the past few years (see, for example SEBPC[..]), and much of this is likely to applicable to the next set of legacy problems.

It is seductive to think that current technology developments, such as components, middleware, enterprise computing and so on will provide the ultimate answer to all problems, and once software applications are expressed in this form, there will be no more legacy software. Experience acquired over the past 40 years shows this is extremely naïve. The ‘end of history’ scenario has proved to be entirely false on a number of occasions, and not just in software engineering (at the end of the nineteenth century, it was assumed that Physics had been completed, and only a few loose ends remained to be tidied up!). It is safe to predict that in 20 years, software engineering will change in ways which we cannot imagine now, and we shall have to work out how to cope with what now is the latest technology, but will become tomorrow’s legacy. In other words, the software legacy problem is *enduring*.

Much effort has been expended over the past fifteen years in technology solutions to legacy systems. Up to now we have avoided mentioning the terms *reverse engineering*, *re-engineering* etc. These topics have led to much interesting research, almost none of which has been exploited industrially in significant ways. This is clear

from the lack of empirical and practice papers in the literature, and the absence of a significant tools and service vendor market. It is time to explore this in more detail.

Keith - there is a market in gateways, see paper by Olsem (citation in our previous paper) YES I AGREE WHAT SHOULDS WE ADD HERE>??????

The terminology lacks crisp definitions, and there is confusion and overlap between the terms available (for example, processes, behaviours and states are confused). In the section on servicing, we carefully avoided introducing new terminology, and concentrated on the need for better methods for program comprehension, supported by code improvement to help. Furthermore, the techniques are often used to refer only to executable legacy code. For many organizations, the data is the strategic asset, rather than the code. Solutions to legacy software may be expensive, but are achievable. But coping with organizational data may be of much higher priority.

Whatever the inadequacies of the terminology, the aims of reverse engineering would seem to be reasonable: to recapture high level design and architectural information about the software, so that it can be re-implemented, or at least maintained more easily. The evidence is that the techniques can help program comprehension, but have not been successful at 're-engineering' systems (for example, extracting high level design and re-implementing that). The existing legacy system may now be the *only* source of information about the organisation's business rules. Research should make these tasks easier. But there are several flaws to this argument:

1. It is assumed that if only we used modern technology in place of the existing legacy, maintenance would be much cheaper/easier. It has been argued earlier that the maintenance implications for many new technologies are not understood. Sneed [..] has found that companies will not spend resource on reverse engineering to new technology on the basis that maintenance costs are smaller.

2. It may seem obvious to designers that having high level design/architectural knowledge with traceability through to high and low level design helps maintenance, but there is little empirical evidence that this helps typical maintenance staff who have responsibility for only part of the system. To them, tactical knowledge of (for example) impact and ripple may be much more important (see Section 3). Pragmatically, if several representations of software exist (e.g. documentation, design, source code), it is only the source code that is maintained, and the other representations become inconsistent and are no longer trusted.

3. It is not clear how much of the design is now relevant; many business rules may have been superceded. So the outcome may be of little *value*.

4. Reverse engineering technology (above the code transformation level) is not fully automatable, and often requires large amounts of time from highly skilled software engineers and domain experts.

5. For many components, the source codes may not be available, and determining their behaviour may be very hard (i.e. expensive).

This may be summarized as: re-engineering is a high risk, high cost, labour-intensive activity with business benefits that are not clear. The staged model offers a clear explanation: the move from initial development to evolution is straightforward because the team knowledge is retained by the expert staff. The move from evolution to servicing is irreversible, because the architecture has degraded, but especially the corresponding human skills have been irredeemably lost. Recovering technology alone is insufficient; and most research has ignored addressing the staff expertise.

It is not surprising that the preferred current solution is *wrapping* legacy software (the behavioural comprehension may be 'encapsulated' in the form of an object interface). Our analysis puts such software (especially large, monolithic software) firmly within the servicing stage – the gap between its capabilities and business needs has become too great. This in turn provides a simple definition of legacy software – software which is in the servicing stage.

This suggests that a new research landscape is urgently needed in the broad field titled 'reverse engineering' to address current problems of current concern, and the problems that will surely arise for the next generation of component-based legacy systems. A more promising research line of attack is based on a two-pronged approach:

1. To explore *multidisciplinary* solutions in terms of both a technical and a business/organisational solution [HEND00 - SABA], in which the many stakeholders in the process have a contribution.
2. To generalize the problem to one of *migration* from the current (legacy) state of affairs to the desired position. This certainly includes data and objects, not just code.

One of the difficulties is that we do not know now what the future software systems will look like. This means that exploring migration solutions, when the target is not clear, is difficult. It is to be expected that the problem is raised from addressing code (where much existing research is concentrated) to addressing components in distributed systems. Input from industry to forming the landscape is likely to be important. Research solutions which can be taken up quickly will be required – solutions which take ten years will be obsolete.

5 Emergent organizations: evolution for the new millennium

The research landscape that has been drawn is mainly a projection of current trends. It has envisaged software development remaining largely as it is now, even though the technology may change substantially. So a system is developed, released to the market place, is evolved though a series of releases typically months apart, and then drops into servicing when it is no longer a strategic product.

This will probably be valid for the tightly constrained traditional software mentioned below. However, it is clear that time-to-market for software has become the top priority for many business applications. For example, a finance house may create a new financial product; it must be implemented and launched within 24 hours; and then has a life of only two more days. Release intervals of years or months need

instead to be days or weeks. The aim of this section is to propose a far more radical and far reaching agenda for research, which will place software evolution center-stage. As noted in the introduction, this section is based on the results of a process which deliberately set out to produce long term views of the future of software. These were certainly not restricted to technical or engineering issues. The process [BRER99] was as follows:

- A group of senior software engineering academics and industrialists met regularly to explore and frame visions of the future of software in terms of a ten year horizon
- A multidisciplinary 'scenario planning' workshop was held, attended by a range of senior user discipline experts as well as software engineers. The disciplines included (for example) civil engineering, law, psychology, business and medicine. The software engineering vision was presented, but then the discipline experts considerably amplified and extended this from a user-oriented perspective.

In [BRER99], the ten year view of software is presented, based on the above process; this steps back from a detailed technology focus and incorporates the views of experts across a wide range of disciplines. It also presents a process for thinking about both shorter-term research (in this chapter, section 2 and 3), and long term research (section 4). The research developed four scenarios for software; the scenarios are grouped under four headings:

- how software and society will interact
- how software will be used
- how software will behave
- how software will be developed

This work led directly to the vision presented below.

One of the main technological conclusions reached was that the 'level of abstraction' of software engineering will continue to rise. Ten years ago, this used to be source code. Now it is components, glue and middleware. For many users, technology is not the main problem, and it is likely to become a progressively smaller problem as standard solutions are bought in from technology vendors. Instead, the focus of research will change to the interface of the software with business. Current software is completely dominated by a technology-focused viewpoint. Although technology vendors provide much baseline technology, people have not even begun to use it effectively. This may partly be due to the awkwardness of the technology in just making things happen and partly because it has an IT-focus model of operation rather than user-oriented models. Over the next ten years, a radical shift to different, user-oriented view of software is inevitable. Much of the foundation of computing is now in place, and we need to explore the empowering nature of modern computing and how this can best be realised.

Software engineering and evolution research and practice are still largely influenced by an era when the boundaries of a problem domain were well-defined and subject to relatively little change. Software production models predominate and have been extremely successful where the application system is tightly defined (e.g. real-time

systems, embedded systems, core database managers, etc.). Complexity in these systems is usually in algorithms and real-time control. Thus through careful process design and attention to these bounded areas of complexity, software changes can be regarded as relatively straightforward, even though they may require deep domain-specific knowledge. Evolution, through discrete system releases, occurs at relatively widely spaced intervals, typically many months, representing discontinuity between versions. A good example is the space shuttle on-board flight system. In strategic terms, research is aimed at producing better (cheaper, more dependable, scaleable) solutions to well understood processes. Many of these activities have reached maturity, such that standards are now being defined [ISO95, IEEE93].

A long-standing problem with software is the *supply-industry* dominated view of "*software as a product*" in which software components are engineered into system solutions. Whilst this approach works well for systems with well-defined boundaries of concern, such as embedded systems, it breaks down for applications where system boundaries are not fixed and are subject to *constant urgent change*. These applications are typically found in *emergent organisations*- "organisations in a state of continual process change, never arriving, always in transition"- such as the new e-businesses or traditional companies who continually need to reinvent themselves to maintain competitive advantage. For emergent organisations, software is often difficult to change at the required rate and has high costs of ownership (such as extra unwanted facilities, steep learning curve, frequent upgrades).

In future, a *user-demand* led view of software will predominate which will result in software being provided as a service. Already some suppliers are making software available on central servers on a pay-per-use basis, but this is only a change to the delivery mechanism. More fundamental will be a change in the way the software itself is constructed. In future, rather than software components being developed and 'bound' together to form a single, rigid solution, systems will be developed as a 'federation' of services which are only bound together at the point of execution. This will enable alternative software components to be substituted between each use of a system, allowing much finer-grained flexibility.

An analogy is making an international telephone call. The caller does own the means of production, but simply pays for the use of a range of third party facilities. Furthermore, when a call is made to the same number over a period of time, the telecommunications operator will route the call in different ways on each occasion in order to optimise cost, network traffic and performance. The caller gets the same service, i.e. is connected to the dialed number, but the provision of the service may change on each call [SERV99].

There is clearly a shift in the software engineering discipline. With the advent of the PC, IT has moved from an exciting ‘new’ technology to one which is all pervasive, ‘a PC on every desk’. The grand research challenges are also now much larger and more complex.

The central *research problem* to be tackled is the inability to change software easily and quickly, and the consequent *cost of ownership*. This problem constrains business enterprise and is the predominant cause of user dissatisfaction with IT. The problem arises because software is *product-oriented*, irrespective of whether it is purchased,

leased or outsourced. Beyond relatively simple configurability, at the point of delivery software is monolithic, which brings many undesirable features such as unwanted functionality, upgrades and difficulty in responding to rapid business change. The Y2K problem is a recent, but by no means final example.

Fundamentally, many organisations now buy in standard technology solutions from suppliers, and technology research is not a central business issue. The 'level of concern' has risen considerably, to the interaction between IT and the business. This necessarily includes many non-technological aspects such as organisational theory, psychology, legal implications, socio-technical aspects etc. Software evolution is set to become much more *interdisciplinary*, and much more concerned with end-user domains in business.

It may be that so called 'emergent models' based on complexity theory may offer an explanation for the phenomenon of software evolution. In these models, relatively simple and small sets of rules can generate highly complex behaviours. There is plenty of scope for highly speculative research in this area.

6 Conclusions

We started by expressing the view that much more empirical knowledge about software maintenance and evolution is needed, including process, organization and human aspects. A novel model of the complete software life-cycle, called the staged model, has been presented. This has been a good vehicle for describing the research landscape in the field. During initial development, the main need is to ensure that subsequent evolution can be achieved easily. This is as much an organizational issue as a technological problem, since much of the expertise to design good architecture is a property of human ability and flair, as well as understanding and representing the architectures themselves, even when the application is not really innovative. In other words, the foundations are laid for a successful evolution phase. The service stage tends to operate at a lower level of abstraction, and there is much scope for improvement to program comprehension and program improvement technologies, especially for component-based distributed systems.

We have identified the area of reverse engineering as one which has not been widely exploited industrially (except in the narrow interpretation of program comprehension), and it now needs a new landscape for research. The field needs to take a broader perspective, of migration, in particular to anticipate the next phase of legacy problems, which are expected to be much greater than those experienced so far. Migration must have a destination, and there is much scope for research to address this.

Finally, we have summarized a long-term radical view of software evolution, based on a service model not a product model. Such a radical view is designed to meet the expected needs of emergent organizations, who are predominantly the type found in e-business and for whom rapid, urgent change is a fact of life. This is motivated by the recognition that currently software has a very strong technology focus for users, and has to become far more responsive to user needs and requirements.

Several major themes have thus emerged. Software evolution needs to be addressed as a business issue as well as a technology issue, and therefore is fundamentally interdisciplinary. To make progress we need to understand what evolution is as well as how to undertake it. Strategically, progress in software architectures is crucial, so that we can extend and adapt functional and non-functional behaviour without destroying the integrity of the architecture in order to respond to unexpected new user requirements.

The ability to change and evolve software easily, quickly and reliably is a 'grand challenge' within software engineering. Change is intrinsic to software; it is one of the benefits, and it is naïve to expect that evolution will not be needed in the future. Incremental improvements are likely to bring general modest industrial benefit in a number of areas. However, improvements of orders of magnitude are going to need radically different ways of thinking about the problem. Too much focus at present is on the technology, not on the end-user. Solutions are going to be essential to meet the needs of businesses where change is constant and urgent. So we can expect software evolution is be positioned at the centre of software engineering.

Acknowledegements

K H Bennett would like to thank the EPSRC and Leverhulme Trust for financial support, particularly through the Systems Engineering for Business Process Change programme. Thanks are due to many colleagues, especially Magnus Ramage and Paul Layzell (who collaborated on parts of section 4), as well as Malcolm Munro, David Budgen, Pearl Brereton, Linda Macaulay, David Griffiths and Peter Henderson. Many ideas for the long term scenario were formed in the British Telecom funded DICE project.

V T Rajlich would like to acknowledge the support from Ford Motor Co. and also NSF research grant # 9803876. These research issues were discussed with Norman Wilde, Franz Lehner, and many others.

==========

BENN95 *Bennett K. H.* ***Legacy Systems: Coping with success****. IEEE Software vol. 12, no. 1, pp19 – 23, Jan. 1995.*

[Benn99] Bennett, K.H., Rajlich, V.T., A new perspective on software evolution: the staged model, submitted.

BRAN94 Brand S. How Buildings learn. Phoenix Ltd. ISBN 0 75380 0500, 1994

BRER99 Brereton O. P., Budgen D., Bennett K. H., Munro M., Layzell P. J., Macauley L. A., Griffiths D. & Stannett C. The future of software: defining the research agenda. Comm. ACM. Dec. 1999.

BURC97 *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.*

CUSU97 *Cusumano M. A.* & Selby R. W. **Microsoft Secrets** *HarperCollins, 1997, ISBN: 0006387780*

HOLT94 ***The architecture of Open VME.*** *ICL publication ref. 55480001, from ICL, Cavendish Rd., Stevenage, Herts, UK SG1 2DY, 1994*

IEEE93 IEEE Std. 1219: Standard for Software Maintenance. Los Alamitos CA., USA. IEEE Computer Society Press, 1993.

ISO95 Int. Standards Organisation. ISO12207 Information technology – Software life cycle processes. Geneva, Switzerland, 1995

KNIG99 C. Knight, M.Munro. Comprehension with[in] Virtual Environment Visualisations.
Proceedings of the IEEE 7th International Workshop on Program Comprehension, *Pittsburgh, PA, May 5-7, 1999, pp4-11.*

Lehman M. M. On understanding Laws, evolution and conversation in the large program lifecycle. Journal of Software & Systems, vol. 1, pp. 213 – 221, 1980

LEHM85 *Lehman M. M.* ***Program evolution****. Academic Press, London. 1985*

LEHM98 Lehman MM and Ramil JF***, Feedback, Evolution And Software Technology - Some Results from the FEAST Project***, Keynote Lecture, Proc. 11th Int. Conf. on Software Engineering and its Application, Vol. 1, Paris, 8 -10 Dec. 1998, pp. 1 – 12

LEHN91 *Lehner F***. Software lifecycle management based on a phase distinction method***. Microprocessing and Microprogramming, vol. 32 (North Holland), pp. 603 – 608, 1991.*

McDER99 McDermid J. and Bennett K. H. Software Engineering research in the UK: a critical appraisal. IEE Software, vol no August 1999

SERV99 Further information available at www.service-oriented.com.

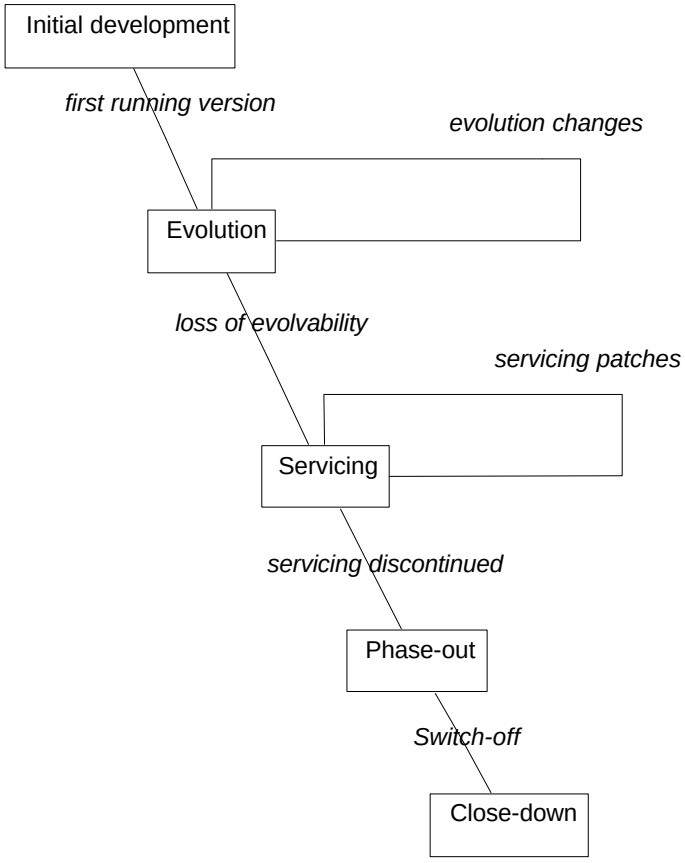

Figure 1. The simple staged model

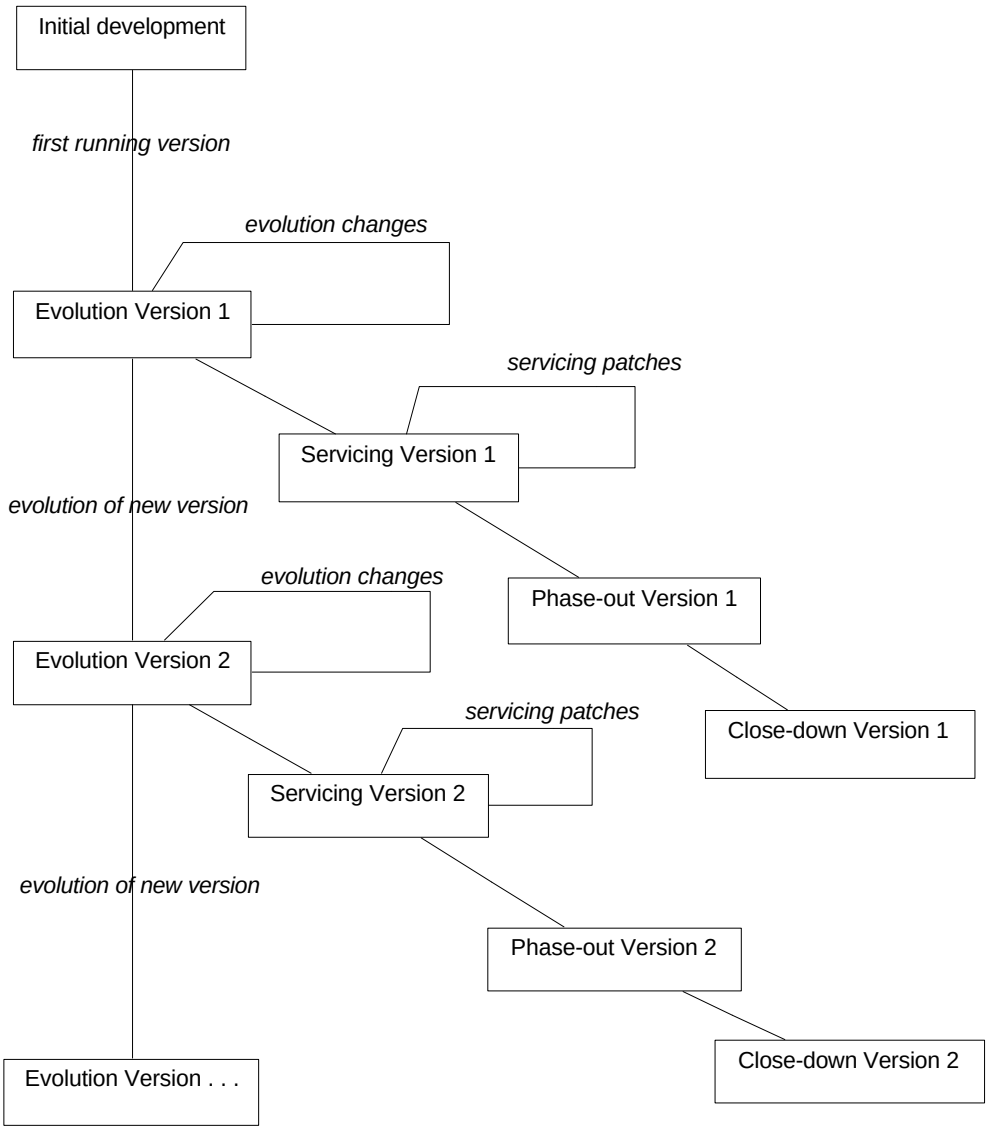

Figure 2. The versioned staged model