The staged model of the software lifecycle: A new perspective on software 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
vtr@cs.wayne.edu

Abstract

In the conventional view of the software life cycle, software is produced, delivered to the user, and then enters a maintenance stage. Maintenance is the more expensive and extensive activity, and many surveys exist that analyze it in terms of constituent activities such as corrective, perfective, adaptive maintenance, etc. In this paper, we suggest a different view that partitions the conventional maintenance phase in a more useful, relevant and constructive way.

We retain initial development, but then propose an explicit evolution stage. Next is a service stage, consisting of simple tactical activities. Later still, the software moves to a phase-out stage and finally to a close-down. 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.

This perspective, by adopting both technical and business roles, is useful in planning and stimulates a set of research issues.

Key words

Software life cycle, software maintenance, software stages, software evolution, code decay, software servicing, legacy, phase-out, close-down.

Introduction

What is software maintenance? Is it different from software evolution? Why isn't software designed to be easier to maintain? 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 term "software maintenance" 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 [IEEE90] as essentially 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.

In a very influential study, Lientz and Swanson [LIEN80] undertook a survey, in which they analyzed maintenance in terms of activities such as:

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

This categorization has been reproduced in many software engineering textbooks and papers, and the study has been repeated in different application domains, in other countries and over a period of 20 years. However, the basic analysis has remained essentially unchanged and is based on the assumption that the maintenance phase is uniform over time in terms of the activities undertaken, the process and tools used, and the business consequences.

The major contribution of this paper is to propose that 'maintenance' is not a single uniform phase, but is comprised of several distinct stages, each with a different technical and business perspective. This new perspective is explained in the paper and is called the *staged model*. We shall restrict our use of the term 'software maintenance' to historical discussions.

2. The stages

Software undergoes several distinctive stages during its life, see Figure 1:

- **Initial development -** the first functioning version of the system is developed.
- **Evolution -** the engineers extend the capabilities and functionality of the system to meet needs of its users, possibly in major ways.
- **Servicing** - the software is subjected to minor defect repairs and very simple changes in function.
- **Phase out** - no more servicing is being undertaken, and the owners seek to generate revenue from the use for as long as possible.
- **Close down** - the software is withdrawn from the market, and any users directed to a replacement system if this exists.

The first stage is *initial development*, where the first version of software is developed from scratch, to satisfy the initial requirements. This stage has been well described in software engineering literature and there are very many methods, tools, and textbooks that address this stage. From the point of view of the future stages, several important foundations are laid during this stage.

The first such foundation is the expertise of the software team. The initial development is the stage during which the team still learns facts about the domain and the problem, no matter how much previous experience was accumulated. This expertise is an indispensable value that will make future evolution possible.

Another important foundation is the architecture of the system. By the architecture we mean the components that comprise the system, their interactions, and properties (functionality, efficiency, etc.) The architecture will either facilitate or hinder the changes that will occur during the evolution. It will either withstand those changes or break down under their impact.

Evolution is the next stage. If initial development was successful, the software enters the evolution stage, where an iterative addition, modification, or deletion of software functionality (program features) takes place. This is partly the result of the learning process in the software team; 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 learning process during the iteration [CUSU97]. Another reason for changes are the customers, who may require additional functionality. Market pressures also add to the growth, by a need to match features of the competitor's product. In some domains, such as public sector software, legislative change will force major evolutionary changes, often at short notice, that were never anticipated when the software was first produced. In other domains, changes in business practices or operating environment of software also lead to unanticipated changes.

Software is released to customers during the software evolution stage. In an extreme case, it is released immediately after the initial development, but more often there are several internal iterations of software evolution that address the most glaring deficiencies so that the released system has a stable fault rate. The time of the release is based on both technical and business considerations where the managers 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 is an arbitrary milestone.

For software to be evolved easily, it has to have an appropriate architecture, and the software team has to have the necessary expertise. When either architectural integrity or the expertise of the architects is missing, the software is no longer easily evolved, and it enters the *servicing stage*, that has been alternatively called saturation [LEHN91], aging software, decayed software, and legacy. During this stage, it is difficult and expensive to make changes, and hence the changes are usually limited to the minimum. The software is often treated as a black box and the changes are implemented as wrappers by modifications of inputs and outputs from the old software. Only limited changes 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.

In our opinion, the Y2k problem was caused by the fact that many 'legacy' systems are in the servicing stage, and although Y2k rectification would be a reasonably easy change during the evolution stage, it is a hard or very hard change during the servicing stage. 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.

The next stage of software life is *phase out*, also called decline [LEHN91]. During this stage, no more servicing is being undertaken, and its owners seek to generate revenue (or benefit) for its use for as long as possible. The software may be still in use, but because the change requests are no longer honored, it is becoming increasingly outdated. The users of the software must work around the known deficiencies of the system more and more often. It may be difficult to return from here to the stage of servicing, because of growing backlog of unsatisfied change requests.

Finally during *close down* the software is shut down, and users are directed to a replacement system if this exists. There may be residual responsibilities such as retention of the source codes, legal liability, etc. This may seem to be a trivial stage, but for some application types (e.g. outsourced software), the management of software assets towards the end of a contract becomes a crucial matter and has received very little attention in the literature. Management of key organizational data is crucial for some organizations, and can dominate decisions about the software. As a system moves through phase-out to close down, migration of the data to a new system must be carefully planned and organized.

In many instances, software is dealt with through a variant of the staged model, called the *versioned stage model*, see Figure 2. In it, the evolution process is the backbone of the model. At certain intervals, a version of the software is completed and released to the customers. At that point, the evolution of the program continues and eventually produces the next version, while the released version is no longer evolved, but only serviced. Many organizations use a naming scheme such as <product>.<version>.<release>, for example MSDOS Version 6 Release 22.
The version refers to strategic changes during evolution, and the release refers to servicing patches.

3. Stage transitions

During both initial development and evolution, the *staff expertise* is critical. The staff must understand the domain of the problem, the solutions of the problems within the domain, the properties of the program including the software architecture, location of domain concepts within the code, software engineering and computer science concepts, the coding conventions, etc. This expertise allows the staff to evolve the program, i.e. to implement substantial changes.

During the servicing stage, the staff expertise is much lower and it does not allow big changes in the program. The expertise may be limited to the view of software as a black box, i.e. it is limited to the inputs and outputs from the software.

There is a further loss of the expertise during the phase-out stage. During this stage, the expertise is limited to the knowledge how to execute the program and no changes are made during that stage.

The *software architecture* also changes its characteristics from one stage to another. During the initial development, the architecture of software is established. The architecture represents a significant commitment, and it will determine the future ease of evolution. In the projects we studied, the architecture changed only very little and hence the architectural commitments made during initial development are decisive factors for the future of the software.

As the changes accumulate, the architecture can lose its original lucidity and integrity. There may be episodes of restructuring where the architecture is being partially rebuilt in order to facilitate future evolution.

In the stage of servicing, the architecture becomes out of step with the needs of evolution and becomes an obstacle, limiting the scope of possible changes. Wrapping is a common process in this stage and it further damages the architecture, making it cryptic and very hard to understand. Where code changes are undertaken, they need to be very tactical and have minimal impact on other components. When the deterioration reaches a certain point, the architecture is no longer serviceable and the phase-out stage is the only option.

The *software decay* can be explained as a positive feedback between loss of staff expertise, and loss of coherence in the program architecture. As the system architecture degrades, there is a stronger demand for greater and more sophisticated expertise to recognize and exploit the core design, to satisfy the main architectural constraints, and possibly to try to improve the architectural design where it has degraded too much. The real expert can understand when a change is tactical, and where it has profound effects on the architecture. The expert is able to spot danger signals. It is almost impossible to document or codify such expertise; it is almost always *tacit*.

However, as a project ages, the tendency is for such strategic expertise to be lost, and for staff with *less* expertise to join the project. Unfortunately, this is exactly the opposite of what is needed to keep the project architecture under technical control. We have positive feedback (with less and less expertise leading to more and more architectural degradation), and the system quickly lurches into the service stage. There is another awkward property of positive feedback systems; it is extremely difficult to reverse this type of behavior.

Re-engineering is an attempt to reverse the decay, but it is a slow and expensive process with many risks involved [OLS98]. Much research (including that undertaken by the authors) has tended to support code-level rejuvenation such as clone removal, but has not addressed the reversal from servicing back to evolution. This explains why the current solution to such problems is wrapping: the software is frozen and wrapped

as a black box. Usually, the aim is gradually to transfer functionality from the wrapped system to a new component or subsystem, so the wrapped system can be phased out.

4. Evidence and support

Our new perspective was derived from published case studies and from personal practical involvement with industrial and commercial software projects, some of them unpublished for proprietary reasons. They are summarized in this section.

Lehner [LEHN91] has provided empirical evidence that the activities and their frequencies change during the lifecycle of a system.

The inevitability of the evolution stage has been well documented in the work of Lehman [LEH85], who documented increases in size, complexity, and functionality during evolution. He observed that *E-type software* is part of its changing environment, therefore there can be no complete and consistent set of requirements established at the start of an E-type project. Evolution is a fundamental result, not an undesirable side effect.

Microsoft Corporation uses a development process [CUSU97] where the division between initial development and evolution is not sharp and the technique of using beta releases to gain experience from customers is common. Microsoft tries to avoid 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 enhancement.

The development of the next version is happening while the existing version is still achieving major market success. So 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 (and to do so would be disastrous). The market strategy was 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.

Microsoft does not support old versions of software, which have been phased out, but they do provide transition routes from old to new versions. 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.

VME operating system has been implemented on ICL (and other) machines for the past 30 years or so, and has been written up by Holt [HOLT94]. It tends to follow the classical X.Y release form, where X represents a version 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. Experts with many years of experience evolved the system and were able to sustain the architectural integrity.
2. Each major release is subject to servicing, and eventually that release is phased-out and closed down.
3. Re-engineering is not used from one major release to another. The team expertise and an excellent architecture support evolution of next versions.

Major billing system is 20 year old, 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 the market. 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.

A small security 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 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). It has not been planned in this way, but has occurred 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 analysis sheds little light on this problem. Our view allows each component and connector to be assessed in terms of its stage. This should allow the company to develop a support plan that takes in account the diversity of stages of individual components.

Long-lived defense system was developed initially in Assembler many years ago and needs to be updated continually to reflect changes in the supporting hardware. According to our analysis, it is still being evolved:

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 both the software and hardware understand the architecture and work on the system. The software is being

changed to meet quite radical new requirements. It is free from ad-hoc patches, and consistent documentation is being produced. The experts understand the impact of local changes on global behavior.

3. Conversely, some experts have recently left the organization, and this loss of expertise, accompanied with fresh signs of structural decay, is a symptom of a serious problem. Re-engineering is not considered feasible, partly because of the lack of key expertise. If the decay reaches a certain point, it is likely that the system will have to be developed again from scratch.

A printed circuits program was developed in a department, managed by one of the co-authors. The program was used by the customers within the same institution and was heavily evolved. The original developers were some of the best personnel in the department and were evolving the program.

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 unsuccessfully to transfer the evolution responsibility to less qualified personnel. However all attempts to train new programmers turned out to be unsuccessful, because the new trainees were able to do only very limited tasks and were unable to make strategic changes in the program. This inability to transfer a "maintenance" task was baffling to the manager. In the hindsight, the expertise needed for the 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.

A CAD tool was developed by a car company 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 dependencies are described by equations that constitute a complex network. Whenever a parameter value is changed, an inference algorithm traverses the network and recalculates the values of all dependent parameters.

After the initial implementation, there was a massive stage of evolution where approximately 70% of the current functionality was either radically changed or newly introduced. The evolution was driven mostly by the user requests and all changes were performed quickly in order to make the new functionality available. The original architecture was not conceived for the changes of this magnitude and deteriorated to the point where the requested changes are becoming increasingly difficult. The symptoms of the deterioration include the proliferation of the clones and misplacement of the code into wrong classes.

Because of that, the evolvability of the software has been decreasing and some changes are becoming very hard. For example, the program would greatly benefit from an introduction of a commercially available inferencing component that supports more powerful inferencing algorithms, but the current architecture with misplaced code and clones does not make that change feasible. The changes done to the software have the character of patches that further corrode the architecture. Recently a decision was made to move software into a servicing stage, outsource the servicing, and to stop

all evolutionary changes. The servicing will meet the basic needs of the users, while a new version of the software is going to be developed from scratch. The attempt to re-engineer the old version has been abandoned.

5. Consequences

The stages of our model differ from each other by different goals, staff expertise, processes, measures, methods, tools, and software properties. Stage boundaries exist and can be characterized and recognized. Note that we do not claim such boundaries are abrupt in time, but are sufficiently evident to separate stages.

The transitions from one stage to the next one can happen as a deliberate business decision, or can happen by default or by mistake. The second kind of transition is something that the managers should beware of because of enormous business consequences. They should understand the symptoms of unintended transition so that they can halt or reverse it while there is still a time. Unfortunately, the "software maintenance" is currently considered to be one monolithic phase and often staffed by second rate people, or done by contractors. The managers should beware that these practices may trigger the unintended transitions. It is crucial to retain the highly skilled staff throughout the initial development and the evolution stages, because it is probably impossible to codify and make explicit the tacit knowledge of these experts. Any attempts to return back to previous stages, or to place the same expectations on software as in the previous stage, are expensive or risky or both. Safety/mission critical software should never enter the servicing stage.

The customers of vendor software should be aware of the stage of the software they are purchasing. They should avoid purchasing software in the advanced stage of servicing.

Profit models (or their equivalent benefit models) for the different stages are also very different. During initial development, there is a heavy investment with no return. There is hence a very powerful incentive to ship the product, both to start to generate revenue, and to beat any competition. It is not surprising that at this stage, it is difficult to support practices that lead to better evolution, because they represent higher cost and deferred returns. However the subsequent evolution stage will be expensive and risky following a poor initial development and the managers should carefully evaluate this trade-off.

Once the product is released, the evolution stage will largely be determined by the success in the market place. If the product is successful, there will be strong and urgent pressures to generate new enhancements, and to make up for ill understood requirements. It is crucial during this stage to try to keep the original team together.

At some stage, following the traditional sales curve, revenue will peak and start to fall. Key team members will move to other projects. We argue that this is the point at which the software moves from evolution to servicing. This can be a process of slippage, but it should be planned.

During the servicing stage, the economics are very different. A well-managed service stage will be determined by budget, and the amount of servicing done should be determined by revenue generated. Moving to servicing involves several big changes:

- Only minor corrections, enhancements and preventive work should be undertaken; evolution-size changes should not be attempted.
- The staff does not require the same level of expertise and much work can be done with partial software knowledge.
- The process should be stable, well understood and mature.
- Cost prediction should be easy.

It is thus possible at this point to outsource the servicing. Work on improving the code can be analyzed in terms of simple economics: does it reduce costs of servicing?

Once a product has moved from evolution to servicing, management should see this as an irreversible process, as the business case for trying to reverse it is hard. Of course, a major software system may represent a large asset, which the company wishes in some way to reuse. If the software as a whole is valuable, then the case is strong to wrap it, which unsurprisingly is the current favored solution. If it is the internals of the software that is valuable, it needs senior personnel to understand this, and this should be done, or prepared for, at evolution time, not servicing.

One of the major concerns of the community is 'what to do with legacy software?' [BENN95]. Our analysis puts such software firmly within the servicing stage – the gap between its capabilities and business needs has become too great. Techniques based around program comprehension are key, and the obvious management solution is to outsource. Furthermore, there is only one direction for such software to go – towards phase out.

6. **Previous work**

Our approach been influenced by Pfleeger's work [PFLE00] on evidence in software engineering. In her terms, we are basing our perspective on tangible evidence and testimonial evidence.

We also have been influenced by [HAND94]. This work has been built on the work of population biologists who have modeled competition and predator-prey situations. The *sigmoid curve* has been the model of the environmental and social situations including marketing. When a successful product is introduced to market, growing sales generate increasing revenue. In time, sales reach a maximum, and then tail off. Eventually, the product is withdrawn. Any organization must plan for successor products well before the sales curve reaches a maximum, in order to sustain revenue. Applying this to software, evolution occurs when sales are buoyant, market demand is strong, and revenue is good. As sales fall off, the business solution is to move the software into its servicing phase. However, it is important to start work on the *next* version during the buoyant period; Handy stresses that if this is left too late (i.e. in the service phase), the market position cannot be recovered.

At first sight, our contribution would seem to be re-inventing a waterfall lifecycle model for the maintenance phase; much of the work on lifecycle models (e.g. Boehm's spiral model [BOEH88]) have stressed the iterative nature of software

development. Of course, the waterfall model is based on the idea of completion of technical deliverables at the end of each stage, and it is widely accepted that this is not viable. In contrast, our perspective proposes iterations, but they are very different at the early and at the late stages of the lifecycle.

A formal model of evolution and an example of evolution of a small program (appointment notebook) appears in [RAJ00].

7. Conclusions

We have described a *staged* model for software process with five largely non-overlapping stages: initial development, evolution, servicing, phase-out and close-down. Our perspective involves the following facts:

- Each stage has very different technical solutions, processes, staff needs and management activities. For example, the solutions for evolution and for servicing are very different.
- A key issue is the nature of the stage changes and boundaries. 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.
- Design for evolution is a topic of central importance. During the evolution, we need to permit high flexibility because we cannot predict how user requirements evolve.

There are a number of topics left open within this broad perspective – for example, cost estimation, validation, methods, tools, measures, re-engineering techniques, etc.; all these aspects are have not been investigated within the context of the stages. We hope our perspective may stimulate such work.

The tendency for modern software is to build it from components. The components may include a variety of parts – COTS, legacy components, custom built components, etc. The parts are *integrated* by glue of various types. Each component will individually be at its own stage of its lifecycle. Our perspective offers a way of reasoning about such systems and how and why they should be supported.

We feel that this is likely to be a major benefit of our approach. Current monolithic legacy systems have proved intractable, but they are at least often homogenous. The next generation of legacy systems, with their use of large, distributed, multi-sourced components, is likely to be much more difficult to address. We need to reason about them before they reach the legacy phase.

References

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

BOEH88 *Boehm B. W.* **A spiral model of software development and enhancement**. *IEEE Computer, May 1988, pp. 61 – 72.*

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

HAND94 *Handy C.* **The empty raincoat**. *Arrow Books, ISBN 0099301253, 1994.*

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

IEEE90 **IEEE Standard Glossary of software engineering terminology**, standard IEEE Std 610.12-1990, **IEEE Software engineering - IEEE standards collection** New York : IEEE, Edition 1994 ISBN/ISSN 155937442X

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

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

LIEN80 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*

OLS98 *Olsem, M.R.* **An Incremental Approach to Software Systems Re-engineering**, *Software Maintenace: Research and Practice 10, 1998, 181-202.*

PFLE00 *Pfleeger S. L., Menezes W***. Technology transfer: marketing technology to software practitioners,** *IEEE Software,* Jan. 2000, 27 – 33.

RAJ00 *Rajlich V.* **Modeling Software Evolution by Evolving Interoperation Graphs**, to be published in *Annals of Software Engineering*, Vol. 9, 2000.

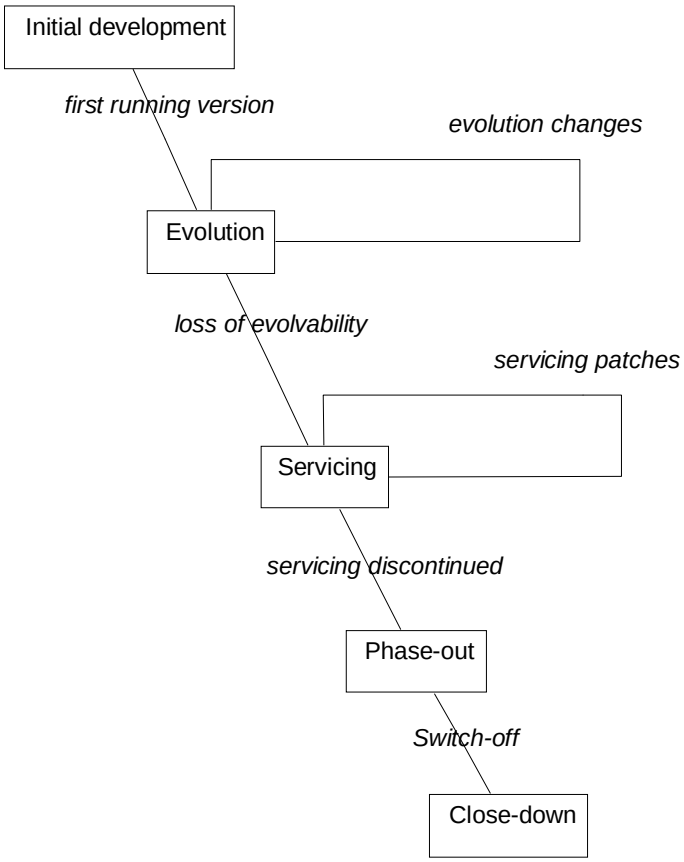

Figure 1. The simple staged model

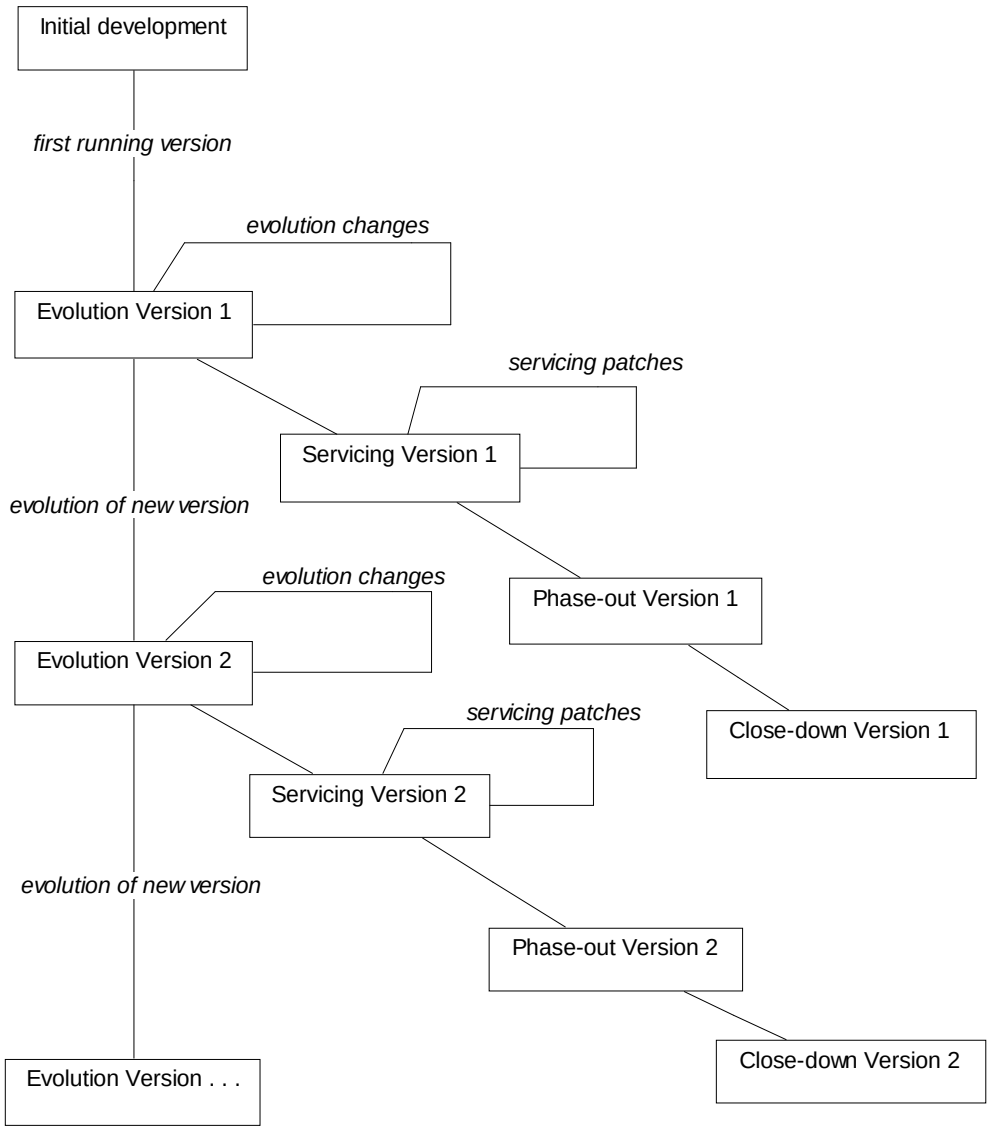

Figure 2. The versioned staged model