A Staged Model for the Software Life Cycle

While the industry still considers post-delivery work as simply software maintenance, the process actually falls into stages, and both management and developers can benefit by understanding them.

Václav T. Rajlich
Wayne State University

Keith H. Bennett
University of Durham

Software engineers have traditionally considered any work after initial delivery as simply *software maintenance*. Some researchers have divided this work into various tasks, including making changes to functionality (perfective), changing the environment (adaptive), correcting errors (corrective), and making improvements to avoid future problems (preventive),[1] but still most have considered maintenance basically uniform over time.

Because software development has changed considerably since its early days, this approach no longer suffices. We describe a new view of the software life cycle in which maintenance is actually a series of distinct stages, each with different activities, tools, and business consequences. Both business and engineering can benefit from understanding these stages and their transitions.

SOFTWARE STAGES

As Figure 1 shows, according to our model, the software life cycle consists of five distinct stages:

- *Initial development.* Engineers develop the system's first functioning version.
- *Evolution.* Engineers extend the capabilities and functionality of the system to meet user needs, possibly in major ways.
- *Servicing.* Engineers make minor defect repairs and simple functional changes.
- *Phaseout.* The company decides not to undertake any more servicing, seeking to generate revenue from the system as long as possible.
- *Closedown.* The company withdraws the system from the market and directs users to a replacement system, if one exists.

A variation of this process is the *versioned staged model,* shown in Figure 2, the backbone of which is the evolution stage. At certain intervals, a company completes a version of its software and releases it to customers. Evolution continues, with the company eventually releasing another version and only servicing the previous version. Many companies use a naming scheme like <product>.<version>.<release>—for example, MSDOS Version 6 Release 22—for this process. The version refers to strategic changes during evolution, and the release refers to servicing patches.

Our work has been influenced by Franz Lehner,[2] who provided empirical evidence that activities and their frequency change during a system's life cycle. Manny Lehman[3] documented the inevitability of the evolution stage, demonstrating increases in size, complexity, and functionality during evolution.

At first glance we seem to be simply reinventing the waterfall life cycle model. However, the waterfall model calls for the completion of technical deliverables at the end of each stage, which is widely considered impractical.[4] We propose making iterations, but in our model of software change and evolution, the iterations are very different in the early and late stages of the life cycle.[5]

Initial development

During initial development, engineers build the software from scratch to satisfy initial requirements. This stage is well documented using numerous well-known tools and methods. From the point of view of future iterations, this stage lays two important foundations:

- *Software team expertise.* During initial development, the team adds significantly to its knowledge of the domain and the problem. This expertise is critical for future evolution.

0018-9162/00/$10.00 © 2000 IEEE

- *System architecture*. The system components, their interactions, and their properties, such as functionality and efficiency, may aid or hinder changes during evolution.

Evolution

If initial development is successful, the software enters the evolution stage, when iterative changes, modifications, and deletions to functionality occur. Evolution partly results from the learning process. As Michael Cusumano and Richard Shelby noted,[6] features may change 30 percent or more as a direct result of learning during an iteration. Customer demands for additional functionality and competitive pressures also cause evolution. In some domains, evolution may respond to legislative action or to changes in business practice or operating environment.

Sometimes companies release software immediately after initial development, but most often the software is released during evolution after the software has gone through several internal iterations to address glaring deficiencies and ensure a stable fault rate. The release date is also based on sometimes-conflicting technical and business considerations, such as time to market, time to delivery, software stability, and fault rates. The release can also occur in several steps, including alpha and beta releases. Therefore the release—the traditional boundary between software development and software maintenance—can be a blurred and somewhat arbitrary milestone.

Servicing

To evolve easily, software must have both an appropriate architecture and a skilled development team. When these are lacking, the software enters the servicing or saturation stage when it is considered aging,[2] decayed, or legacy. During this stage, changes are both difficult and expensive, so developers minimize them or do them as wrappers, which are simply modifications to inputs and outputs, leaving the old software untouched. Still, each change further degrades the architecture, pushing it deeper into servicing.

Mission-critical software should never enter the servicing stage in which requests for changes cannot be honored. The Y2K problem is a good example. It would have been relatively easy to resolve if software had been in the evolution stage, but most legacy systems were in the servicing stage, when such significant changes are very difficult. The problem caught many managers unprepared because they did not understand the difference between evolution and servicing.

Phaseout

During the phaseout or decline stage, the company undertakes no more servicing and tries to generate revenue, or other benefits, from the unchanged software

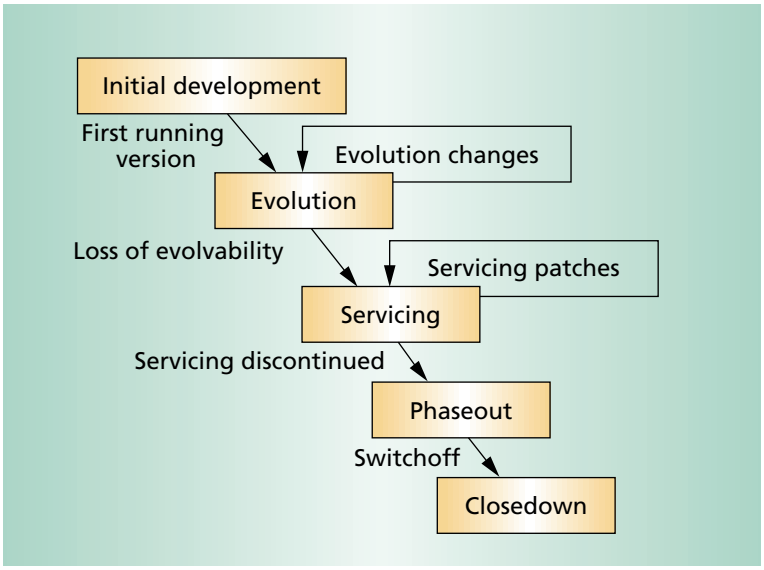

Figure 1. The simple staged model for the software life cycle consists of five distinct stages.

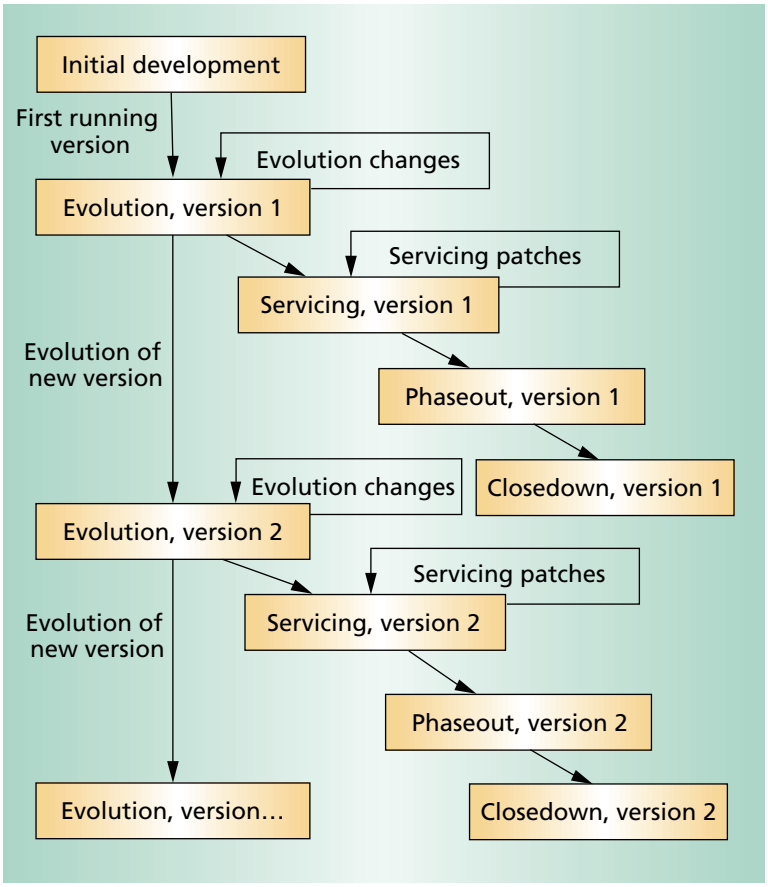

Figure 2. The versioned staged model for the software life cycle emphasizes the evolutionary nature of software development.

The release—the traditional boundary between software development and software maintenance—can be a blurred and somewhat arbitrary milestone.

as long as possible.[2] People may still use the software, but with no changes made, it becomes increasingly outdated, and users must work around its deficiencies. It is difficult now to return to the previous servicing stage because of the growing backlog of change requests.

Closedown

During the final closedown stage, the company shuts down the software and directs users to a replacement system, if one exists. Still, the company may have residual responsibilities, such as source code retention and legal liability, which are particularly important in areas such as outsourced software, where there are contractual obligations. In some companies, management of key organizational data is crucial, dominating software decisions. As a system moves from phaseout to closedown, managers must carefully plan and organize migration of data to a new system.

STAGE CHARACTERISTICS

Several characteristics of software and the development team change substantially from one stage to another, including staff expertise, software architecture, software decay, and economics.

Staff expertise

Staff expertise is critical during both initial development and evolution. The staff must understand the domain, solutions to domain problems, program properties, including software architecture, concept location within the code, and basic computer science principles and coding conventions. With this expertise, the staff can evolve the program, implementing substantial changes.

Staff expertise is less important during servicing because big program changes are not allowed. Expertise may be limited to inputs and outputs, with the software considered a black box. Expertise is even less important during phaseout, when it can be limited to simply knowing how to execute the program.

Software architecture

Software architecture can also change from one stage to another. During initial development, the staff establishes the architecture, which represents a significant commitment—one that determines the ease of future evolution. As changes accumulate, however, the architecture loses its original lucidity and integrity, sometimes requiring restructuring in which designers partially rebuild the architecture to facilitate future evolution.

During servicing, the architecture falls out of step with the needs of evolution and becomes an obstacle, limiting the scope of possible changes. Wrapping then becomes common, further damaging the architecture by making it cryptic and hard to understand. When code changes are made, they need to be very tactical to minimize the impact on other components. Deterioration finally reaches a point where the architecture is no longer serviceable, and phaseout is the only option.

Software decay

Software decay is the positive feedback between the loss of staff expertise and the loss of architecture coherence. As architecture degrades, there is a greater need for expertise to recognize and exploit the core design, to satisfy the main architectural constraints, and possibly to significantly improve the architectural design. Only experts can fully understand when a change is tactical, when it will have profound effects on the architecture, and when it will cause serious problems. This expertise is nearly impossible to document; it is almost always tacit.

However, as a project ages, staff with this expertise tend to leave, and they are replaced by people having less expertise—exactly the opposite of what is needed to keep project architecture under technical control. The positive feedback, then, is less expertise leading to more degradation, with the system quickly lurching into the service stage.

Reengineering is an attempt to reverse decay, but it is slow and expensive, with many risks.[7] Many researchers have considered code-level rejuvenation, but they have not successfully addressed returning to evolution from servicing. Therefore, again, the current solution to such problems is wrapping, with the goal of eventually transferring functionality to a new component or subsystem and then phasing out the wrapped system.

Economics

Our model has been influenced by the work of population biologists who apply competition and predator-prey relationship models to business.[8] According to their projections, a product's sales follow a sigmoid curve in which they reach a maximum and then trail off, eventually forcing the product's withdrawal. In order to sustain revenue, a company must plan successor products well before the sales curve reaches a maximum.

Initial development involves heavy investment with no return. Managers want to quickly ship the product, both to generate revenue and to beat any competition. They often discourage practices that could allow better evolution because they represent higher costs and deferred returns, but this investment can make evolution easier and less expensive.

With software, evolution occurs when sales are buoyant, market demand is strong, and revenue is

good. As sales fall off, businesses should move software into its servicing phase. However, starting work on the next version during the buoyant phase is critical. If this happens too late, companies cannot recover their market position.

After release, the product's success will largely determine the evolution stage. Success will bring strong and urgent pressures to generate new enhancements and make up for poorly understood requirements. Keeping the original team together is crucial during evolution.

At some point, following the sigmoid curve, revenues will peak and start to fall, causing key team members to move to other projects. It's at this point, we believe, that software moves from evolution to servicing. Economics now change significantly, with revenues strongly determining the level of work. Servicing has these features:

- Staff undertakes only minor corrections, enhancements, and preventive work, not attempting evolution-size changes.
- Staff does not require the same expertise, handling most work with partial software knowledge.
- Work is stable, well understood, and mature. Cost prediction is relatively easy.

Servicing can therefore be outsourced. Decisions on improving the code are based simply on whether work will reduce servicing costs.

Because of difficulties in returning to evolution, management must view servicing as irreversible. Companies that view their software as a valuable asset will seek to reuse it, usually through wrapping. However, if the software's internals are valuable, senior personnel need to understand this and prepare for changes during evolution, not servicing.

Legacy software is a significant concern.[9] Because the gap between business needs and capabilities is so great, our analysis puts legacy software firmly in the servicing stage. Management can apply techniques only so far as the software is understood, but companies often outsource this work. Legacy software, though, is still destined for phaseout.

CASE STUDIES

In developing our model, we reviewed published case studies and investigated a number of industrial and commercial software projects, some of which remain anonymous for proprietary reasons.

Microsoft software

Microsoft's development process makes no sharp division between initial development and evolution, and the company often releases beta versions to gain experience from customers.[6] Microsoft tries to avoid the traditional maintenance phase, realizing that its large user base makes this logistically impossible. Its object code patches—or service packs—fix only serious errors and do not try to enhance the software.

Microsoft starts developing the next version while the existing version is achieving major market success. For example, Microsoft did not wait until Windows 95 sales began to decline before developing Windows 98, which would have been disastrous. Microsoft bases its market strategy on a rich and expanding set of features, so Windows 98 replaced 95, which never underwent evolution.

Microsoft does not support old versions but simply phases them out and provides transition routes to new versions. Interestingly, Microsoft has not felt the need for substantial code documentation, indicating that its design teams are stable enough to retain tacit knowledge. Evolution, then, is Microsoft's main activity, with relatively little effort directed to servicing.

If the software's internals are valuable, senior personnel need to understand this and prepare for changes during evolution, not servicing.

VME operating system

For more than 30 years, the virtual machine environment (VME) operating system has run on International Computers Ltd. and similar machines.[10] The company has tended to use the classical X.Y release form, with X representing the version and Y representing minor, or servicing, changes. As with Microsoft, major releases tend to represent market-led developments, incorporating new or better facilities.

The VME operating system is remarkable for the length of time it has kept its original architectural attributes despite a huge evolution in facilities. It's unlikely that original source code from the early 1970s remains, yet VME clearly preserves its architectural integrity. We therefore make these conclusions:

- VME made a heavy investment in initial development, resulting in a meticulous architectural design. Experts with many years of experience evolved the system and were able to sustain architectural integrity.
- Each major release is subject to servicing and eventual phaseout and closedown.
- The company does not reengineer from one major release to another but relies on team expertise and an excellent architecture to support evolution of next versions.

Major billing system

One of the companies we investigated offers a major billing system that is 20 years old, continues to generate revenue, and is of strategic importance. However, in recent years the marketplace has changed, and the billing system can no longer keep up.

Evolution requires expertise that is equivalent to or perhaps even greater than the expertise required to create a program from scratch.

Our analysis showed that this system has slid from evolution into servicing without management realizing it. Key designers have left, the architectural integrity has been lost, changes will take far too long to implement, and revalidation would be a nightmare. It is a classic legacy system, and the only solution is to replace it, which will be very expensive.

Embedded software

We looked at a small security company with a niche market in specialized security devices. The company's products use rapidly changing hardware peripherals, and the company must always be concerned with product sophistication in order to keep ahead of the competition. The devices use embedded software, mostly based on Microsoft products, with vendor-supplied off-the-shelf components, locally written components, some legacy code, and glue written in C, C++, and Basic. Such a conglomeration was not planned; it just happened.

The software is a constant source of problems: New components must work with legacy code. Powerful components are linked with low-level code. Support for locally written components is becoming difficult. From our perspective, the company has a software system with some parts in initial development, some in evolution, some in servicing, and others ready for phaseout. There is no sustained architectural design.

Considering all of this, traditional software maintenance offers little help, but viewing components and connectors according to our model should allow the company to develop a support plan that takes into account the diversity of stages.

Long-lived defense system

The long-lived defense system we examined was initially developed in Assembler many years ago and now needs continual updating to reflect changes in supporting hardware. According to our analysis, the system is still evolving. We identified these characteristics:

- The software is and will continue to be mission critical to the organization. Software failure would be a disaster.
- Many experts with in-depth knowledge of both software and hardware understand the architecture and work on the system. They are changing it to meet radical new requirements, freeing it of ad hoc patches, and producing consistent documentation. These experts understand the impact of local changes on global behavior.
- However, the recent departure of some experts, along with fresh signs of structural decay, indicates a serious problem. The organization does not consider reengineering feasible, partly because of the lack of key expertise; however, if decay continues, the company will have to develop the system again from scratch.

Printed circuits program

A department one of us managed developed a printed circuits program for users in the same institution. The original programmers, some of the best personnel in the department, were evolving the program. When the department experienced a backlog of projects requiring high expertise, the manager, who did not then understand the difference between evolution and servicing, tried to transfer evolution to less-qualified personnel.

However, all attempts to train the new programmers failed, and they were unable to handle the needed strategic program changes. The inability to transfer this "maintenance" task baffled the manager, but in hindsight it's clear that evolution requires expertise that is equivalent to or perhaps even greater than the expertise required to create a program from scratch. Ultimately, assigning new projects to new programmers and leaving the printed circuit program to the experienced developers proved more cost effective.

CAD tool

An automobile company developed a CAD tool to design mechanical components such as engines and transmissions. The tool was implemented in C++, with every mechanical component modeled as a C++ class. Equations described the component dependencies, creating a complex network. When users changed a parameter value, an inference algorithm traversed the network and recalculated values for all dependent parameters.

After initial implementation, an evolution stage followed, radically changing or introducing new features to about 70 percent of the original functionality. This resulted mostly from user requests, and developers responded quickly to add the needed functionality. The original architecture, however, had never been designed for changes of such magnitude. The system showed typical symptoms of deterioration, including proliferation of clones, misplacement of code, and reliance on patches.

The program's ability to evolve has continued to decrease, and now making some changes is very difficult. For example, the program would benefit by adding a commercially available inferencing component that supports more-powerful inferencing algorithms, but the clones and misplaced code make such an addition infeasible.

This company recently decided to move the software into a servicing stage, outsource that work, and

stop all evolutionary changes. The servicing will meet basic user needs while the company develops a new version from scratch.

CONSEQUENCES

Our model's stages exhibit different goals, staff expertise, processes, measures, methods, tools, and software properties. There are recognizable boundaries between the stages, but these boundaries are not necessarily abrupt. It is important to understand stage elements and transitions.

Transitions from one stage to the next can result from deliberate business decisions or by default or mistake. Managers must be conscious of the enormous business consequences of unintentional transitions and be aware of their symptoms so they can halt or reverse them while there is still time. Managers should also understand that attempts to return to previous stages or to deal with software as if it's in a previous stage can be both expensive and risky.

In particular, managers should watch for situations in which "software maintenance" is considered one homogeneous phase and handed over to second-rate programmers or outside contractors. Retaining highly skilled staff from initial development through evolution is crucial because it is often impossible to codify and make explicit the tacit knowledge of these experts.

Customers should know what stage software is in and should explicitly ask for that information before buying it. They should avoid any software in the advanced servicing stage because the software is unlikely to evolve with the user's needs, and closedown is probably near.

We make these conclusions about the five stages—initial development, evolution, servicing, phaseout, and closedown—in our software process model:

- Each stage has very different technical solutions, processes, staff needs, and management activities.
- Managers who have a better understanding of stage transitions, their characteristics, and information flow across them can plan better.
- Keeping systems within a particular stage for as long as possible is important.
- Developers must design systems to allow high flexibility during evolution because managers cannot predict what new user requirements will arise.

The tendency now in software development is to build from components—including COTS, legacy components, and custom-built components—all integrated with software glue. This means that each component may be at its own individual stage during the system's lifetime. Our perspective provides a way for developers to understand these increasingly complicated systems before they reach the legacy phase. ✷

References

1. B. Lientz and E.B. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations,* Addison-Wesley, Reading, Mass. 1980.
2. F. Lehner, "Software Life Cycle Management Based on a Method for Phase Distinction," *Euromicro J.*, Aug. 1991, pp. 603-608.
3. M.M. Lehman, *Program Evolution,* Academic Press, London, 1985.
4. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.
5. V. Rajlich, "Modeling Software Evolution by Evolving Interoperation Graphs," *Ann. Software Eng.*, Vol. 9, 2000, pp. 235-248.
6. M.A. Cusumano and R.W. Selby, *Microsoft Secrets,* Simon & Schuster, New York, 1998.
7. M.R. Olsem, "An Incremental Approach to Software Systems Re-engineering," *Software Maintenance: Research and Practice*, May/June 1998, pp. 181-202.
8. C. Handy, *The Empty Raincoat,* Arrow Books, London, 1994.
9. K.H. Bennett, "Legacy Systems: Coping with Success," *IEEE Software,* Jan. 1995, pp. 19-23.
10. N. Holt, *The Architecture of Open VME,* ICL, Herts, UK, 1994.

Václav T. Rajlich *is a full professor and former chair in the Department of Computer Science at Wayne State University. His current research interests include software change, evolution, comprehension, and maintenance. He received a PhD in mathematics from Case Western Reserve University. Contact him at vtr@cs.wayne.edu.*

Keith H. Bennett *is a full professor and a former chair in the Department of Computer Science at the University of Durham. His current research interest is new software architectures that support evolution. He received a PhD in computer science from the University of Manchester. He is a chartered engineer and a fellow of the British Computer Society and the IEE. Contact him at keith.bennett@durham.ac.uk.*

XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX