Overview of Software Maintenance and Evolution

Jeff Offutt
George Mason University

January 2018

This overview of software maintenance is drawn from multiple sources. We are at a relatively strange period in software engineering where maintenance and evolution activities account for much, if not most, of software costs, yet most of our understanding is based on studies that are decades out of date.

Sommerville defines software maintenance as: "When the transition from development to evolution is not seamless, the process of changing the software after delivery is often called software maintenance" [1].

More generally, maintenance involves modifying a program after it has been put into use. We usually do not expect maintenance to involve major changes to the system's architecture. Rather, changes are made by modifying existing components and adding new components to the system. From a programmer's perspective, the key issue is that the programmer must understand the program and its structure.

Maintenance is important because software is crucial to company's success and because software is very complicated. In today's world, most of the software budget is devoted to modifying existing software rather than developing new software

Any successful software product is in use much longer than in development, making maintenance even more crucial. If software does not continue to adapt to changes in needs and environment, it becomes progressively less useful. Changes are inevitable, and occur for several reasons:

- New requirements emerge when the software is used
- The business environment changes
- Faults must be repaired
- New computers or equipment are added to the system
- Growing user base makes the performance or reliability insufficient.

Some changes are due to the tight coupling with the environment. When software is installed, it changes that environment, in effect changing the software requirements

Myths Related to Software Maintenance and Evolution

In practice, many software engineers and managers have falsely been convinced of several things that are simply not true. Some of these myths are explained and contradicted below.

Myth: "We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?"

Reality: The book of standards may very well exist, but is it used? In many cases, the answers to the following questions are "no."

1. Are software practitioners aware of its existence?
2. Does it reflect modern software engineering practice?
3. Is it complete?
4. Is it streamlined to improve time to delivery while still maintaining a focus on quality?

Myth: If we get behind schedule, we can add more programmers and catch up.
Reality: Software development is not a mechanistic process like manufacturing. As Brooks said: "adding people to a late software project makes it later."
Myth: If I outsource the software project to a third party, I can just relax and let that firm build it.
Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.
Myth: General objectives are enough to start programming—we can fill in the details later.
Reality: A poor up-front definition is a major cause of failed software efforts. If you don't know what you want at the beginning, you won't get what you want.
Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.
Reality: It is true that software requirements change, but the impact of a change depends on when it is introduced.
Myth: Once we write the program and get it to work, our job is done
Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer.
Myth: Until I get the program "running" I have no way to assess its quality
Reality: One of the most effective software quality assurance mechanisms can be applied from the beginning of a project—the formal technical review. Software reviews are more effective than testing for finding certain classes of software defects.
Myth: The only deliverable work product for a successful project is the working program
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.
Myth: Software engineering will make us create voluminous and unnecessary documentation and will always slow us down
Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Traditional Types of Maintenance

Maintenance has traditionally been divided into four types. The percentages given in textbooks are quite similar, but are based on studies from the early 1980s. It seems unlikely the numbers still hold, so take these with a grain of salt. They are all we have.

1. Perfective maintenance (50%): Enhancements where new operations and refinements are added to existing functions.
2. Adaptive maintenance (25%): Modifying the application to meet new operational circumstances.
3. Corrective maintenance (21%): Eliminating errors in the program's functionality.
4. Preventive maintenance (4%): Modifying a program to improve its future maintainability.

Some authors will consider emergency maintenance as being a type of corrective maintenance that is not scheduled.

A major change in the last 20 years in the way software is maintained is that companies often release an intentionally partial version, then add to it over time. It is not clear to me whether this is properly considered to be perfective, adaptive, or something else. But it is clear that the term evolution is more appropriate than maintenance. This is possible when software is easy to update. For example, software in our mobile devices can be updated every day, as opposed to software in submarines, which can normally only be updated when the submarine visits port. Indeed, this is a major driver behind the Internet of Things movement. Fixing a software problem in my washing machine requires an expensive visit by a technician to replace a circuit board, whereas if my washing machine is on the internet, the company can download new software for almost no cost.

Costs of Software Maintenance and Evolution

Modifying software, no matter how or when, is difficult and costly. When software and hardware is tightly integrated, software is also often looked at as the easiest part to change. But just because the manufacturing cost is so slow does not mean the design and implementation cost of making changes is free.

Large software programs are extremely complicated, and very difficult to understand. Yet they must be understood before they can be changed. Many changes are rushed, so the modifications are often poorly designed and implemented To make things worse, changing the software almost invariably injects new faults that must be repaired later.

Sommerville [1] claims that 90% of all software costs are related to software evolution. Even if that's overstated, few would doubt that more than half of software costs are accrued after first deployment. The costs are due to both technical and non-technical factors. Most engineers agree that the cost of making the same type of change goes up as the software ages. This is due to many factors, including loss of memory about the initial design and construction, changes in technologies and languages, and the difficulty of understanding previous (often sloppy) changes.

Some common **technical** cost factors are:

1. **Complexity of code**: Complex control and logic structure is hard to understand and therefore hard to change.
2. **Changes in programming languages**: If the code to change is written in a language that is no longer commonly used, it is hard to change. Even harder is if new code has to be written in a different language.
3. **Changes in software infrastructure**: If the underlying software, middleware, or libraries have changed, then the programmers have to understand how the software to change interacts with them, which is very difficult.
4. **Quality of code formatting and style**: Messes always create maintenance debt. Perhaps the worst problem is when names are poorly chosen.
5. **Program age and structure**: As programs age, changes start to degrade their structure, making them harder to understand and change.

Some common **human** cost factors are:

1. **Team stability**: Maintenance costs are greater if the original developers are not available.
2. **Contractual responsibility**: If the original developers were not expected to be responsible for future changes, they had very little incentive to design for change.
3. **Staff skills**: Maintenance tasks are sometimes given to entry-level engineers, summer interns, or poorly educated low-skilled programmer. This means they may take longer and their changes may be less reliable.

Software Maintenance Terms

Most of these definitions are taken from IEEE standards [2] or textbooks (which probably derived them from the IEEE).

- Maintainability: The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment. Maintainability cannot be measured directly, but is affected by factors such as age, size, programming language, architecture and design, and documentation and formatting. Maintainability is also quite subjective, since the engineer's ability to understand the software is heavily influenced by prior knowledge and experience with other programs.
- Ripple Effect: Changes in one software location can impact other components. Ripple effects cannot be fully understood with static analysis of the source, so dynamic analysis must be used. That is, we have to run the program.
- Impact Analysis: The process of identifying potential consequences of a change in terms of how the change will effect the rest of the system [3]. This is a difficult analysis that is done to estimate the cost of a change, to choose which fault repair is most cost-effective, to plan for regression testing, or to understand what resources will be needed to make the change.
- Traceability: The degree to which a relationship can be established between two or more products of the development process, such as the requirements and code, or design documentation and tests [3].
- Legacy systems: A software system that is still in use but the development team is no longer available. Legacy systems have often been inherited, is still valuable, and is very hard to change.

Reality Check

As said above, most of the available literature on maintenance and regression is 30 years old! The IEEE standards are from 1990, and only changed modestly from their 1983 versions. The major textbooks have been updated regularly since the 1980s, the information on maintenance has not changed much.

The literature is also conflicted over the use of maintenance and evolution. A possible distinction is:

- Software Maintenance: The activities required to keep a software system operational and responsive after it is deployed.
- Software Evolution: A continuous change from a lesser, simpler, or worse state to a higher or better state.

I have also heard them distinguished as software maintenance being about fixing the software so that it is close to working as intended (adaptive, corrective, and preventive), whereas evolution is about changing its intended behavior (perfective).

Lehman's Laws of Software Evolution

Before concluding, any overview of maintenance or evolution must have a nod to Lehman's laws. They were first proposed around 1980, and still seem to be relevant today.

1. **Law of Continuing Change**: Software that is used in a real-world environment must change or become less and less useful in that environment.

2. **Law of Increasing Complexity**: As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon (hence refactoring).
3. **Law of Self Regulation**: Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors are approximately invariant for each system release.
4. **Law of Conservation of Organizational Stability**: Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
5. **Law of Conservation of Familiarity**: Over the lifetime of a system, the incremental system change in each release is approximately constant.
6. **The Law of Continuing Growth**: The functionality offered by systems has to continually increase to maintain user satisfaction.
7. **The Law of Declining Quality**: The quality of a system will appear to be declining unless it is adapted to changes in its operational environment.
8. **The Feedback System Law**: Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

References

[1] Sommerville, Software Engineering, edition 10, Addison-Wesley Publishing Company Inc., 2015

[2] IEEE, Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, New York, 1990

[3] Wikipedia, Change impact analysis, https://en.wikipedia.org/wiki/Change_impact_analysis, accessed January 2018

[4] Wikipedia, Lehman's laws of software evolution, https://en.wikipedia.org/wiki/Lehman%27s_laws accessed January 2018