Report: "Software evolution and maintenance" Seminar: "Advanced Topics in Software Engineering"

Norman Weinert

19.07.2019

Contents

1 Introduction

The part of software evolution and maintenance in a programs life-cycle is crucial. Up to 70% of a software project's costs can be related to software maintenance alone[1]. The following graphic shall give an example:

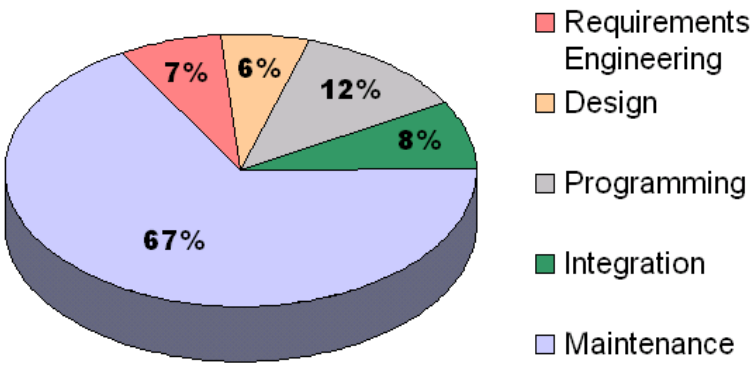

Figure 1: Example of the distribution of a program's costs [2]

Figure 1 shows that only about 10% of the costs are related to actual programming, while 67% are software maintenance costs. The value of optimizing maintenance procedures and reducing the costs, as a result, is very high. Finding defects early can already help in decreasing maintenance costs. The cost of finding mistakes during the stating of the program's requirements are near to none while finding defects after the program has been distributed to the customers is many times higher[3].
Software evolution and maintenance can also help with tackling common after release questions:

- How and when to prepare the next release?

- Can we with an acceptable risk currently include major changes at the current stage?

- What overall strategy shall we follow?

As a result, the importance of understanding and optimizing the process of software evolution and maintenance is crucial to a program's success and in this report, we shall go over the basic definitions, common behaviors with related examples and the process of analyzing a program's data to form a strategy.

2 Definitions

2.1 Software maintenance

Software maintenance focusses on keeping the software running[4]. In contrast to traditional maintenance activities though it aims to deviate the program from the standard[1]. Take a car mechanic as an example. In case of a defect, they will try to turn the broken car back to its original state. Software does not change though and the defects are in its original state as a result. So in order to fix the defects, the program needs to be changed away from the original form. Usually, four categories of software maintenance are mentioned[4]:

- **Corrective maintenance** focusses on fixing errors.

- **Adaptive maintenance** tries to adapt the program to a different environment.

- **Perfective maintenance** wants to improve aspects like the program's performance.

- **Preventive maintenance** attempts to avoid possible problems that could arise in the future

All the different changes resulting from software maintenance improve the program's internal structure and are most likely unnoticeable to the user[4].

2.2 Software evolution

Software evolution is focussed on changing the software‘s overall design. It wants to innovate rather than preserve the original structure and as a result, a new better-adapted system should evolve from the old one. The introduced changes usually also affect the external structure and are visible to the user[4].

3 Laws of software evolution

The laws of software evolution were defined by M.Lehmann and try to capture the process of software evolution as a whole. In this section, we will cover the different kinds of software categories and examine the eight laws of software evolution one by one. In the end, an example from the real world will be checked if it follows those laws.

3.1 Categories of software

In general, we differentiate between three different categories of software[1]:

At first, we've got S-type-programs. S-type-programs have an exact specification on their functionality and they provide exactly that. An example would be a program that calculates the greatest common divider of two numbers.

Then we've got P-type-programs. They implement certain procedures that fully determine the functionality of the program. An example for this would be a chess-program that implements the rules/procedures of a chess game.

At last, there are E-type-programs. E-type-programs try to mimic a real-world activity. How it achieves that depends on the environment. The program's requirements may also vary and it needs to adapt accordingly. For example, we could take a flight-simulator which shall mimic the real-world activity of flying a plane. As new plane models are released the requirements of the program can change so that it has to implement support for these new plane models.

3.2 The eight laws of software evolution

Over the years eight laws of software evolution were defined for legacy E-type-programs. We shall now go over all of them and highlight their core principles[5]:

1.) **Continuing Change**. If a program does not change, it will become less and less satisfactory. So in order to satisfy users the program must continuously be changed.

2.) **Increasing Complexity**. During a program's life-cycle it will become more and more complex, unless direct countermeasures and maintenance to reduce the complexity are performed.

3.) **Self Regulation**. The overall process of software evolution is subject to a self-regulating dynamic. This dynamic also makes the programming process itself self-regulating. As a result, statistically determinable trends and invariances exist.

4.) **Conservation of Organisational Stability**. The global activity rate is statistically invariant during the program's active life.

5.) **Conservation of Familiarity**. The released content of all releases of an evolving program is statistically invariant.

6.) **Continuing Growth**. If a program does not increase its content it will become less and less satisfactory for the user.

7.) **Declining Quality**. The overall quality of a program will seem to decrease unless countermeasures are applied. This includes the adaptation to operational environment changes.

8.) **Feedback System**. The process of software evolution includes multi-level (different program versions), multi-loop (inclusion of code reviews) and multi-agent (can be managers, developers, testers,...) feedback systems to improve itself.

Now we will take a look at an example, as [6] checked to which degree the laws of software evolution apply to the open-source program Mozilla Firefox in contrast to a legacy program for which the laws were originally made for. The following table shows if [6] could confirm Mozilla Firefox's observance of the laws:

Law	Confirmation
Continuing Change	Yes
Increasing Complexity	Yes
Self Regulation	No
Conservation of Organisational Stability	No
Conservation of Familiarity	No
Continuing Growth	Yes
Declining Quality	Yes
Feedback System	No

Table 1: Mozilla Firefox's observance of the laws of software evolution

Table 1 shows that only four of the eight laws could be confirmed in the case of Mozilla Firefox. We will take a closer look at how [6] came to their result in the case of the **Continuing Change** and **Feedback System** law.

In order to confirm the law of **Continuing Change** [6] made use of a graph that shows the number of lines of code over the different versions of Mozilla Firefox. The graph shows that over most of the time the number of lines of code is varying. This indicates changes in the system. Even the later period where no huge changes in numbers were made was deemed as enough to confirm the law.

For the **Feedback System** law, it was stated that Mozilla Firefox did only include multi-level feedback in forms of different versions, but no multi-loop or multi-agent feedback. The main feedback source is the userbase alone and this only satisfies the law partially and as a result, the observance cannot be confirmed.

4 Feedback-driven software evolution and maintenance

The overall process of software evolution is mostly feedback driven[1][5]. As a result, the majority of successful and useful software aims to satisfy the user‘s wishes[7]. There are generally two major types of feedback: Positive and negative feedback.
Positive feedback tends to entail change and growth[5], as it indicates that the software is in a relatively stable state and the risk of introducing new features and doing greater changes to the existing content can be taken. Negative feedback tends to entail a period of stabilization with only minor growth[5]. It indicates that there are defects in the current state of the software and introducing greater changes now could destabilize it even more. As a result, fixes need to be done first before further growth can be undertaken. Reaching this stable enough point is again probably indicated with the incoming of mostly positive feedback.
Knowing this the following graphic introduces the versioned stage model[7] which further underlines the dynamic of positive and negative feedback:

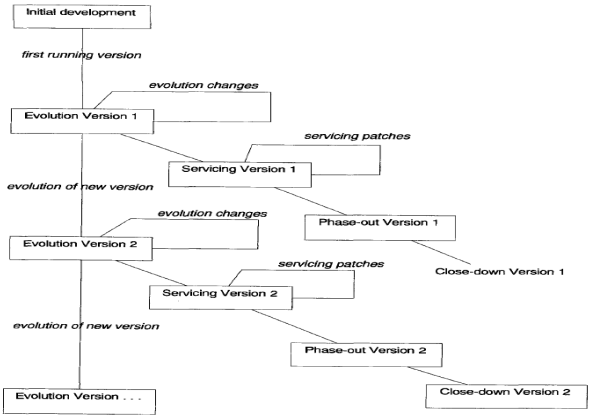

Figure 2: The versioned stage model[7]

Figure 2 shows that after the initial development or the introduction of a new evolution version there will probably be a lot of defects in the program due to the introduction of greater changes and as a result, will most likely lead to negative feedback. Now servicing versions aim to fix these found defects and please the wishes of the customers. After enough defects have been fixed and the program is in an acceptable state, most likely indicated by positive feedback, a phase-out version is released at which point no further servicing versions are being developed. Now the focus lies on preparing the next major evolution version. With the release of the next evolution version, a close-down version for the outdated release is being handed out to directly point users to the next release. At this point, the cycle starts all over again.
We will now look at an example project from the real world to underline this behavior: NeoLemmix, a Lemmings clone based on the original Lemmings game from 1991[8][9]:

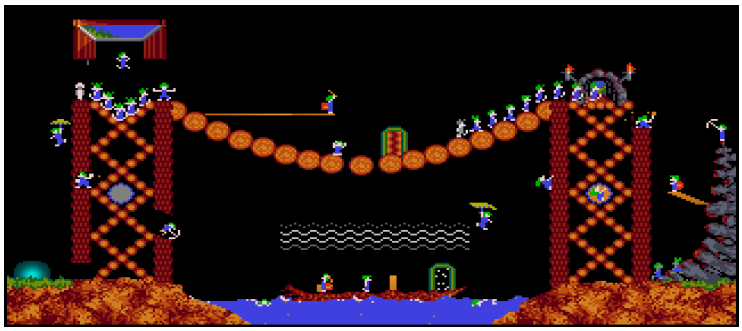

Figure 3: Lemmings clone NeoLemmix[8]

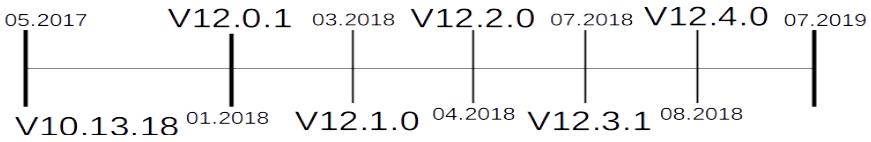

Figure 4: Timeline of the mentioned versions of NeoLemmix

Figure 4 shows a timeline for the different versions of NeoLemmix relevant for our example. V 10.13.18 of NeoLemmix was stable for a longer time period and gained a lot of positive feedback. As a result, the next major released aimed at a big existing problem: A lot of the data was stored in user-unreadable files and in order to be edited and customized they needed to be accessed with additional tools. The next release V 12.0.1 changed the whole program‘s format as a result and stored the data in user-readable files instead. They were easy to swap out and edit and the reliance on additional tools was greatly reduced. With this major release, the next four versions (V 12.1.0, V 12.2.0, V 12.3.1 and V 12.4.0) focussed on addressing the negative feedback that came with the changes. A lot of features were not working as intended and some old features were either not functioning yet or completely culled to reduce complexity. This needed to be addressed first through those servicing versions and the final phase-out version before preparing the next major release[10][11].
V 12.4.0 is very stable again and gains a lot of positive feedback. Following this, the next version aims to focus on growth and innovation by introducing new mechanics and new customization options[12][13]. After this future version is released a new stabilization period is very likely.
This development mimics the dynamic of positive and negative feedback discussed earlier. Also, as described in the versioned stage model, servicing versions (12.1.0, V 12.2.0 and V 12.3.1), phase-out versions (V 10.13.18, V 12.4.0) and major releases (V 12.0.1 and the upcoming version) are indeed present. Only a close-down version pointing at the new major release is missing, but due to the very small development team further technical support for older versions is difficult to maintain and therefore secondary.

5 Software evolution resulting from competition and collaboration

Competition and collaboration have a high impact on software evolution[14]. Following that, we will take a closer look at the implications of these factors and an example highlighting them.
Programs, in general, evolve to satisfy and attract more customers and therefore generating more profit. As a result, they also evolve to either match or surpass the quality of the competition, as users are more likely to use the software with the highest quality. Because of this usually, competition is inevitable, especially for smaller companies as those need to claim their place in the market at the same time[14].
Collaboration is often used to develop greater and/or more complex programs. Without it, bigger projects would otherwise be most likely impossible. Again, this especially counts for smaller companies. Different programs with different strengths can also learn/adopt functionality from each other to improve themselves. This newly adopted functionality can also be improved further, as the other team might have a new viewpoint on the topic.
We will take a look at another example now which highlights this behavior: Lix is a Lemmings-like game based on the original Lemmings game from 1991 just as NeoLemmix:

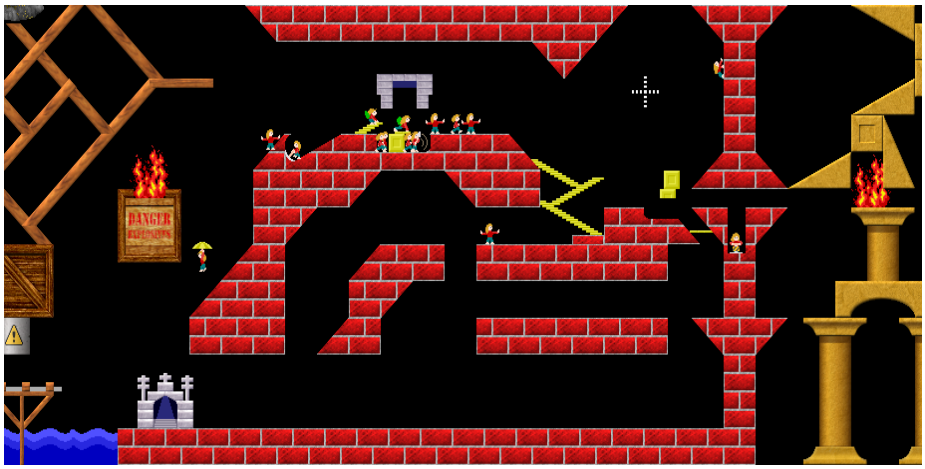

Figure 5: Lemmings-like game Lix[15]

Lix and NeoLemmix have a friendly history of collaboration between each other but are also aiming at the same target group.
A problem in the old Lemmings game from 1991 was that every failed attempt needed to be replayed from scratch, even if the error was right before the end. This resulted in a lot of retries that cost a lot of time. In order to avoid this and focus more on the puzzle-solving side of the game rather than the execution side, NeoLemmix implemented a live rewind feature that enabled the player to rewind actions frame-by-frame or even by several seconds. The feature was well received but very unresponsive for longer levels as rewinding caused the program to internally replay every action the player has done up to the point they want to rewind to[16]. The unresponsiveness surfaced after about three to four minutes into a level.
Lix then implemented an improved version of the feature inspired by NeoLemmix. The improvement was that the program makes internal savestates from time to time so that a fixed point from which to replay from was always nearby. The memory needed for savestates is very low and even in the most extreme cases levels don't go over roughly 30 minutes, in fact, in most cases, they don't go over six minutes. As a result, these internal savestates made rewinds always responsive[17].
Soon after NeoLemmix implemented a similar improved method based on internal savestates to address the performance problem[18]. Both programs are now of higher quality thanks to them collaborating, adapting features from each other and improving those even further by addressing known problems associated with them.

6 Forming and deciding on a strategy

Forming and deciding on a strategy is crucial for avoiding problems and generating a fluent program life-cycle. To form a proper strategy, information to base it on is needed. This can, for example, come from the current program's history, or other similar programs. Viewing the development of key aspects over a time period enables to learn from past mistakes, notice current mistakes and prevent future mistakes in advance. Example of such key aspects would be the number of modules, the number of changed modules, or the release interval[1]. Figuring out which aspects are relevant to the project's strategy can be challenging and is not invariant for different projects. There are many different methods to extract relevant information such as[19]:

- Developer effort and social network analysis. Developer effort focusses on showing which developer did the most work in the different parts of the project so that workforces can be redistributed if suboptimal distributions are discovered. Social network analysis analyses the structure of the social network of the project in order to make the projects team structure more efficient.

- Trend and hotspot analysis tries to highlight parts of the project that are growing quickly or are being changed often to avoid too radical growth and notice important or vulnerable parts of the project.

- Fault and defect prediction analyses the parts of a project where the majority of faults and defects are being discovered. This could be an indicator to make those parts more robust or to restructure them entirely.

Graphs are often used for the visualization of certain aspects of the project‘s history. Now we will go through a few examples of graphs visualizing certain information about an example project covered in [1]. First, we take a look at a graph showing the system size in thousands of modules in relation to the system age in thousands of days:

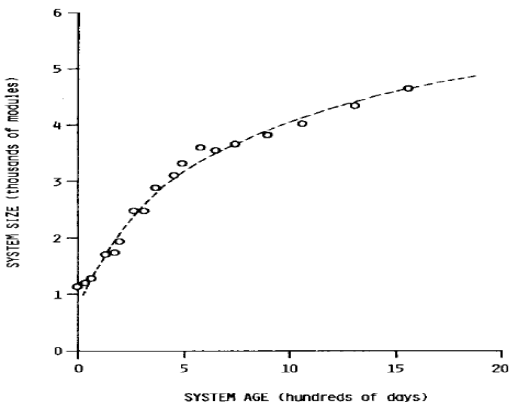

Figure 6: System size in thousands of modules over the system age in thousands of days[1]

In figure 6 every point symbolizes a release version and the graph shows that the system is in a state of continuing growth according to the related law of software evolution. While the growth from version to version seems to be roughly constant, the time period between releases is increasing further and further. This could indicate that the program might be getting closer to the end of its life cycle. Next, we take a look at a graph visualizing the change of the program. It shows the increase in the number of modules for the different release sequence numbers:

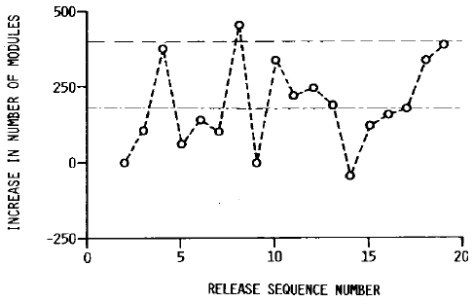

Figure 7: Increase in number of modules over the release sequence number[1]

Figure 7 shows a fluctuating behavior as in some releases a high number of new modules is introduced, while in other releases just a small number is added. This could be indicating periods of stabilizing servicing versions due to negative feedback and innovating evolution versions due to positive feedback, as we discussed earlier. For the last two versions visualized here quite a high number of modules were introduced, so we could predict that the next release will focus on stabilization and maintenance rather than on innovation and evolution.

We see that in order to gain and visualize crucial information to form a fitting strategy we can use graphs and statistics of our own project's history or the history of fitting similar projects. From this information, we can determine if there are critical faults, or any essential features missing. We will now take a look at the planned release 20 for the project the graphs were covering before and try to form a strategy based on our gained knowledge:

TABLE III
RELEASE 20 PLANNED CONTENT

Functional Enhancement

No.	Description	New Mods (NM)	Old Mods Chgd (OMC)	Mods Chgd (NM + OMC)	OMC/NM (IR)
1.	Identified faults (PRE. RLS. 19)	2	380	382	–
2.	Expected faults (RLS. 19)	0	600	600	–
3.	Interactive terminal support (ITS)	750	1783	2533	2.4
4.	Dynamic storage management (DSM)	170	1500	1670	8.8
5.	Remote job entry (RJE)	57	462	519	8.1
6.	New disk support (NDS)	17	124	141	7.3
7.	Batch scheduler improvements (BSI)	3	29	32	9.7
8.	File access system (FAS)	8	74	82	9.3
9.	Paper tape support (PTS)	12	80	92	6.7
10.	Performance improvements	2	157	159	–
		1021	5189	6210	

Figure 8: Planned content for release 20[1]

We can see that for release 20 they want to introduce over 1000 new modules, while also changing over 5000 old ones. Most changes appear to be in the "Interactive terminal support (ITS)" part of the program with 750 new modules introduced and over 1700 modules changed.
As we saw in figure 7, the last two releases already introduced a lot of new modules. Under these circumstances, it is highly recommended to redefine release 20 as a release focussing on stabilization rather than innovation and delay the planned changes to release 21 or even 22, as defining release 21 as a stabilization version in addition to 20 could also be advantageous. So all in all the project is in desperate need of a stabilization period before being ready again for further evolution steps.

7 Conclusions

We saw that feedback is a very important source of information for deciding when to evolve and when to focus on maintenance. Positive feedback most likely leads to change and growth, while negative feedback most likely leads to periods of stabilization and maintenance. Competition can be a big driver of evolution, as programs try to attract more customers and surpass their competitors. Collaboration with other companies can help to manage larger changes and further improve the program. It can also lead to the exchange and improvement of features. We were able to experience these mechanics in our example with the NeoLemmix and Lix projects.
In order to form a fitting strategy for a project, gaining the right information with various methods is key. The information has not to be from the project itself, as other fitting similar projects can be used as well. The right strategy assists in realizing past, present, and future problems, so lessons can be learned and countermeasures can be applied. We formed an example strategy for a project described in [1] where we applied what we learned.
All the mentioned points contribute to minimizing maintenance costs while also helping to provide a high-quality product for customers.

References

[1] Meir M. Lehman:
Programs, Life Cycles, and Laws of Software Evolution 1980

[2] Schach, R.:
Software Engineering, Fourth Edition, McGraw-Hill, Boston, MA, pp. 11. 1999

[3] Tim Menzies, William Nichols, Forrest Shull, LucasLayman:
Are Delayed Issues Harder to Resolve? Revisiting Cost-to-Fixof Defects throughout the Lifecycle 2016

[4] Michael W. Godfrey and Daniel M. German:
The Past, Present, and Future of Software Evolution

[5] M.M.Lehman, J.F.Ramil, P.D.Wernick, D.E.Perry and W.M.Turski:
Metrics and Laws of Software Evolution - The Nineties View 1997

[6] Anita Ganpati, Dr. Arvind Kalia, Dr.Hardeep Singh:
Software Evolution: An Empirical Study of Mozilla Firefox 2012

[7] Keith Bennett and Vaclav Rajlich:
Software Maintenance and Evolution: A Roadmap 2012

[8] *NeoLemmix homepage: https://www.neolemmix.com [Link used:23.05.2019]*

[9] *NeoLemmix manual: https://www.dropbox.com/s/ypjzkjlndbspwlz/NeoLemmix%20Manual.pdf?dl=1 [Link used: 23.05.2019]*

[10] *https://www.lemmingsforums.net/index.php?topic=3686.0 [Link used: 23.05.2019]*

[11] *https://www.neolemmix.com/?page=download_list&program=16 [Link used: 23.05.2019]*

[12] *https://www.lemmingsforums.net/index.php?topic=4079.0 [Link used: 23.05.2019]*

[13] *https://www.lemmingsforums.net/index.php?topic=4188.0 [Link used: 23.05.2019]*

[14] George Valenca, Carina Alves, Virginia Heimann, Slinger Jansen and Sjaak Brinkkemper:
Competition and Collaboration in Requirements Engineering: A Case Study of an Emerging Software Ecosystem 2014

[15] *Lix homepage: http://lixgame.com/ [Link used: 25.05.2019]*

[16] *https://www.lemmingsforums.net/index.php?topic=2368.0 [Link used: 25.05.2019]*

[17] *https://www.lemmingsforums.net/index.php?topic=2241.0 [Link used: 25.05.2019]*

[18] *https://www.lemmingsforums.net/index.php?topic=2417.msg55320#msg55320 [Link used: 25.05.2019]*

[19] Tom Mens, Serge Demeyer:
Software Evolution (2008), Springer-Verlag Berlin Heidelberg