On the Role of Evolvability for Architectural Design

Stephan Bode

Faculty of Computer Science and Automation
Technical University Ilmenau
P.O. Box 100565
98684 Ilmenau
Stephan.Bode@TU-Ilmenau.de

Abstract: Today software systems have to face frequent requests for change during their whole lifetime. It is very important that they can adapt to the frequently changing needs and are flexible for new features in order to remain useful and to conserve business value. This ability of software systems, known as evolvability, does still not gain the attention it deserves. This paper discusses, why evolvability has to be explicitly considered during the design of software architectures and why the current practice with focus on maintainability during software evolution is insufficient. Furthermore, some advice for tackling the problem is given.

1 Introduction

Today a software system usually represents a major investment for an enterprise, which thus has to be profitable. Therefore, if a system is build once, it has to remain usable for a long time. By now it is commonly known, that there is a permanent demand for software changes, because of the technological evolution, the optimization of processes, or because of the integration of existing systems into the development of new software architectures.

At the latest since the publications of Lientz and Swanson (e.g., [LS80]) one fact is known to every software architect and developer: most of the development costs and time do not result from initial development but from software maintenance. However, in the scope of software maintenance, changes frequently have to be performed with low effort and in short time frames. Therefore, changes can lead to deterioration in the structure of the software, which in turn hampers or inhibits further changes. This effect is called architectural decay or architectural drift.

However, a replacement of affected systems by a completely new development and, hence, the prevention of this effect generally is not an acceptable solution. A replacement potentially comes along with a financial risk for the enterprise and mostly is impossible with regard to deadlines or budgets. That is why the software, and especially

its architecture, has to be able to cope with frequent requests for change to permanently stay usable.

Because of the frequent software changes and of the unsuitability to replace existing software systems as explained above, a demand beyond software maintenance is arising to keep the software system in a condition that allows quick and easy changes for the long term. Therefore, it is the author's opinion, that today's practice in software maintenance and to achieve software maintainability are not enough for the architectural design of long-living software systems, and we additionally have to consider *evolvability* as an important quality goal for the architectural design phase.

In the following section the current situation and practice of software maintenance and the development for maintainability is discussed. In section 3 we argue why it is important to consider evolvability as a goal for architectural design and why a distinction between software evolution and software maintenance and maintainability is necessary. Further, we give some hints how to improve architectural design concerning evolvability. The paper concludes with section 4 where also a plan of future actions is suggested.

2 Why considering only maintainability is not enough

In recent years many works focused on software maintenance and maintainability. There are already standards such as the IEEE Standard for Software Maintenance, which defines maintenance as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [IEEE], or the ISO standard 9126 [ISO01], which defines maintainability as a software quality attribute and refines it for clarification into some sub-characteristics: analyzability, changeability, stability, testability, and maintainability compliance.

So, there were already a lot of efforts to describe how to deal with software changes and modifications such as corrections, improvements, or adaptations to changing needs. Rajlich and Bennett described the software life cycle with their *Staged Model* [RB00] (see Figure 1) explicitly considering an evolution and a servicing stage for software maintenance activities, which are separated from initial development. Furthermore, the organization of the maintenance process has been explored [Ap05]. Maintenance activities can either be performed by the developers or by an extra maintenance team [Wa90].

Broy and Deissenboeck et al. [BDP06, De07] even developed an activity-based quality model [WDW08], which aims at a better understanding of maintainability as a software quality attribute. This approach tries to overcome some typical drawbacks of standard quality models. It relates maintenance activities to factors of the maintenance context, like infrastructure, system properties or organization, in a 2-dimensional matrix to derive control mechanisms.

Summing it up, one could argue, that there has been a lot of research on software changes and their handling through software maintenance, which, if it is only realized in practice, already is sufficient altogether to care about requests for changes and modifications of software systems. Moreover, dealing with maintainability in practice is difficult enough and sufficient. A differentiation between maintainability and additionally evolvability as quality attributes is rather of an academic nature than reasonable. The aspect of software evolution and evolutionary changes is sufficiently covered for example by the evolution stage in the staged model (see Figure 1).

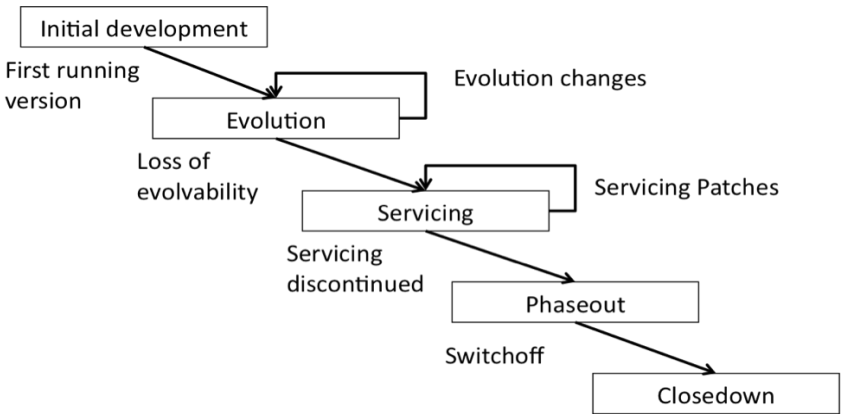

Figure 1: The staged model for the software life cycle according to [RB00]

At first sight these arguments seem convincing, but this is only part of the truth. In reality maintainability and the abovementioned sub-characteristics are rather insufficient. There are further aspects like understandability and program comprehension or traceability of design decisions that have to be considered. Program comprehension, for example, constitutes more than 50% of the effort for maintenance [BR00]. High maintainability prolongs the software's lifetime by lowering development risks, and the abovementioned activity-based model can support it. However, this quality attribute emphasizes the current short-term effort for changes and so does not focus on the long-term upkeep of the software. For example, maintainability can be improved by enhancing code quality but without considerable impact on the software's ability to evolve.

Furthermore, maintenance is focused on relatively small changes due to time and budget restrictions, which can lead to incomplete changes and subsequent errors. Maintenance activities do not necessarily contribute to the maintainability and evolution of a system, because they normally do not consider structural changes. For example, extensions to the software can lead to code duplication, which decreases maintainability and, hence, can lead to architectural decay.

Indeed, architectural decay is an issue [Da08]. If the decay is present, reengineering activities are necessary. Of course, today there are agile methods for software engineering like the famous extreme programming (XP), which try to solve some problems by their simplicity, short release cycles, and by performing refactoring. Nevertheless, these methods are not suitable for large, long-living systems with large development teams, where clear division of tasks is needed based on documentation. There are no sufficient means to describe the architecture and assert it during evolution.

Another point left out so far is the existence of trade-offs between maintainability and other quality attributes. Sure, maintainability normally is not examined on its own, and there are for example the goal-oriented approaches from requirements engineering like the NFR-framework [Ch00], i* [Yu95], or GRL [Am03]. They try to solve conflicts already between different requirements. There are even works, e.g., [FG07, LY01], which want to utilize the goal-oriented approaches for architectural design. However, there are some drawbacks from an architectural point of view. The goal-oriented models are clearly focused on requirements and not on architectural design, so, further means for architecture description are necessary. Moreover, architectural styles and constraints concerning the environment or technical solutions are not considered.

In addition to the previous discussion, in the following section some further arguments are pointed out, why evolvability has to be explicitly considered for architectural design in future.

3 Why considering evolvability is important and necessary

The term software evolution was already introduced in the 1970s when the change problem became apparent with the first large software systems, and it gained more attention in the 1990s. Today, software evolution is an accepted research area in software engineering [MD08]. Evolution according to Lehman's famous *laws of software evolution* [Le80] is targeted at the development of software systems during the whole life cycle, from initial development till closedown. Contrary to revolution it emphasizes the character of stepwise, continuous software changes. In the staged model of Rajlich and Bennett evolution is a stage in the software life cycle, which enables evolutionary changes, separated from the servicing stage, which is reached when maintainability gets lost. Accordingly, evolvability describes an ability to keep maintainability for the long-term.

Getting back to the main thesis, we claim, that it is necessary to distinguish between maintainability and evolvability, as already considered necessary by Breivold et al. in [BCE07]. A discussion on the related terms evolvability, maintainability, evolution, and maintenance can be found in [RB09]. In contrast to maintainability, evolvability considers also evolutionary aspects for software changes like structural changes or architectural integrity. This is reflected in the definition proposed by Breivold et al. [BCE07]: *„Software evolvability is the ability of a software system to adjust to change stimuli, i.e. changes in requirements and technologies that may have impact on the*

software system in terms of software structural and/or functional enhancements, while still taking the architectural integrity into consideration."

Additionally to maintenance activities, evolutionary changes typically comprise structural modifications as well. There are, for example, the integration of new technologies such as wrapping an existing system as a service in a service-oriented architecture, the realization of new quality requirements like an increase in scalability and multilingualism for global operation, or changing an architectural style such as the introduction of a new abstraction layer for persistence. Emphasize is put on the evolution of the architecture and not on keeping the system running. Evolution is not limited to legacy systems, but it is necessary for all software systems that still have a high value for an enterprise. The demand for software evolution and a long lifetime of complex software systems goes much beyond software maintenance, because effort, strategic activities and complexity are considerably more challenging.

For future software architectural design, it is important to establish evolvability as a major quality requirement. On the one hand, only performing maintenance activities does not inevitably mean to increase maintainability or even evolvability, examples are fixing bugs or extending logical conditions. On the other hand, increasing the evolvability of software systems to adapt to future changes also means to improve maintainability as well. To keep a system's evolvability high is the only way to avoid it turning into a legacy system [MM98] (compare the Staged Model).

In order to encounter the challenges in software engineering and especially with software changes today, it is necessary to consider structural as well as process aspects. Evolvability is a quality attribute that includes both. When designing software architectures, means of expression with a high degree of abstraction like domain specific languages or model driven development enable software changes with low effort and so contribute to the evolvability of a system. Means to better support evolvability already discussed in [RB09] are architectural patterns for variability, components models, or plug-in interfaces. Further ways could be the use of generative techniques, which can be helpful for the preparation of changes, or the implementation of techniques associated with product line engineering, which reduce the demand for changes by preparing variants.

Measures for improving process aspects are, for example, software processes considering evolutionary development, or the introduction of traceability for design decisions, which enhances long-term evolution. Besides, a determined consideration of evolvability as a goal during software design leads to the evolution of processes by relating analysis results with adequate activities as for example in the evolvability-oriented development process in [BBR09].

We have to consider evolvability in the context of other quality requirements; hence, it has to be related to them. In doing so, the goal-oriented approaches from requirements engineering [Ch00, Yu95, Am03] can be helpful. Evolvability and further quality requirements can be expressed as goals and related with so-called contribution links according to synergies or conflicts between them. However, only providing such

requirements-oriented goal models for the software architect is not sufficient. Instead, these approaches have to be adapted for the architectural design process to consider architectural styles, patterns, and other technological constraints and solutions.

Moreover, for a better description and understanding of evolvability during architectural design and for a refinement in a goal model, evolvability has to be analyzed and decomposed into sub-characteristics. In contrast to maintainability, which is defined and refined into sub-characteristics in the ISO 9126 for example, there exists no standard quality model for evolvability. Actually, this situation will not change despite the current revision to the new ISO 25010 [ISO]. Nonetheless, there are already works with proposed sub-characteristics [BCE07, BBR09].

However, for architectural design a refinement into sub-characteristics is not enough. Principles of good design, such as separation of concerns, encapsulation, loose coupling, and strong cohesion, and further architectural styles and patterns have to be mapped to the sub-characteristics with an annotation if they are contributing positively or negatively to the quality goals. These principles and patterns again can be mapped to functional and technical solutions for the non-functional (or quality) goals. For these mappings, we propose the concept of the Goal Solution Scheme, which was already discussed for the quality attribute security in [Bo09].

Quality attributes must always be made measurable; however, for evolvability establishing key metrics is an issue worth further research because only limited knowledge is available to date. In [BBR09] and [BR08], for example, feature tangling and scattering are argued as metrics for evolvability. Subramanian and Chung [SC03] discuss process-oriented metrics.

Equally important for an evolutionary development, we do not only have to consider well-established principles, styles, and patterns, but also all maintenance activities and, additionally, reengineering activities. Such reengineering activities are for example code refactorings, which are popular with agile software development methods. However, for the evolvability of software architectures, beyond that, a restructuring of models is even more important.

Now having discussed some arguments for the importance of evolvability for architectural design, in the next section a conclusion about the topic is presented.

Conclusion and future work

This paper provides a discussion, why during the design of software architectures more attention than common has to be put on software systems' ability to evolve over time. It was argued, why only dealing with maintainability and maintenance activities is not enough. The main points stated in favor of evolvability are: firstly, we have to distinguish between maintainability and evolvability as well as between maintenance and evolution because they have a different focus; secondly, we have to consider evolvability among other quality attributes as a major goal for architectural design to strengthen

software system's ability to adapt to frequently changing needs and remain usable; thirdly, we have to further analyze evolvability as a quality goal and its sub-characteristics for a better understanding, to sharpen developers' awareness and for a better support during the design phase.

Beside the argumentation pro evolvability there were also proposed some actions for an improvement of the current situation. The goal-oriented approaches from requirements engineering should be enhanced and utilized for architectural design. This would serve the solution of trade-offs between different quality requirements next to evolvability. Moreover, the concept of the Goal Solution Scheme was suggested as a way to align solutions to the quality requirements. The scheme's mapping of quality goals to principles and solutions guides the developer during the architectural design phase. An important issue is further to reason about proper principles such as separation of concerns or about activities like structural refactoring for reengineering. All together should be integrated in an advanced method for architectural design.

References

[Am03] Introduction to the User Requirements Notation: Learning by Example. Computer Networks, Elsevier North-Holland, Inc., 42(3): 285-301, 2003.

[Ap05] April, A.; Huffman Hayes, J.; Abran, A.; Dumke, R.: Software Maintenance Maturity Model (SM^{MM}): The software maintenance process model. Journal of Software Maintenance and Evolution: Research and Practice, Wiley, 17(3): 197-223, 2005.

[BBR09] Brcina, R.; Bode, S.; Riebisch, M.: Optimization Process for Maintaining Evolvability during Software Evolution. In: Proceedings 16th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009), San Francisco, USA, April 13-16, IEEE, 2009, pp. 196-205.

[BCE07] Breivold, H. P.; Crnkovic, I.; Eriksson, P.: Evaluating Software Evolvability. In (Arts, T. Ed.): Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden (SERPS'07), Göteborg, Sweden, October 24-25, ACM, 2007, pp. 96-103.

[BDP06] Broy, M.; Deissenboeck, F.; Pizka, M.: Demystifying Maintainability. In: Proceedings of the 2006 International Workshop on Software Quality (WoSQ'06), ACM, 2006, pp. 21-26.

[Bo09] Bode, S.; Fischer, A.; Kühnhauser, W.; Riebisch, M.: Software Architectural Design meets Security Engineering. In: Proceedings 16th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009), San Francisco, USA, April 13-16, IEEE, 2009, pp. 109-118.

[BR00] Bennett, K.; Rajlich, V.: Software Maintenance and Evolution: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering (ICSE'00), ACM, 2000, pp. 73-87.

[BR08] Brcina R.; Riebisch, M.: Architecting for Evolvability by Means of Traceability and Features. 4th Intl. ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering, IEEE, 2008. pp. 235-244

[Ch00] Chung, L.; Nixon, B. A.; Yu, E.; Mylopoulus, J.: Non-functional Requirements in Software Engineering. Kluwer Academic Publishers, 2000.

[Da08] Dalgardo, M.: Examples of software architectural decay. Online: http://blog.software-acumen.com/2008/06/05/examples-of-software-architectural-decay/, June 5, 2008, Accessed On: 2009-04-28.

[De07] Deissenboeck, F.; Wagner, S.; Pizka, M.; Teuchert, M.; Girard, J.: An Activity-Based Quality Model for Maintainability. In: Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007), IEEE Computer Society, 2007, pp. 184-193.

[GF07] Grau, G.; Franch, X.: A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures. In: Proceedings of the First European Conference on Software Architecture (ECSA'07), LNCS 4758, Springer, 2007, pp. 139-155.

[IEEE] IEEE Std. 1219-1998, IEEE Standard for Software Maintenance. IEEE Computer Society, 1998.

[ISO] ISO/IEC 25010 Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) Quality model. ISO, under development

[ISO01] ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model. ISO 2001.

[Le80] Lehman, M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE, 68(9): 1060-1076, 1980.

[LS80] Lientz, B. P.; Swanson, E. B.: Software Maintenance Management. Addison-Wesley Longman, 1980.

[LY01] Liu, L.; Yu, E.: From Requirements to Architectural Design – Using Goals and Scenarios. In: From Software Requirements to Architectures Workshop (STRAW 2001), 2001.

[MD08] Mens, T.; Demeyer, S.: Software Evolution. Springer, 2008.

[MM98] Mens, T.; Mens, K.: Assessing the Evolvability of Software Architectures. In: Proceedings of the Workshop on Object-Oriented Technology (ECOOP'98), LNCS 1543, Springer, 1998, pp. 54-55.

[RB00] Rajlich, V.; Bennett, K.: A staged model for the software life cycle. IEEE Computer, 33(7): 66-71, 2000.

[RB09] Riebisch, M.; Bode, S.: Software-Evolvability. Informatik-Spektrum, Springer, 32(4), Aug. 2009. DOI 10.1007/s00287-009-0349-2 (to appear)

[SC03] Subramanian, N.; Chung, L.: Process-Oriented Metrics for Software Architecture Evolvability. In: Proceedings 6th International Workshop on Principles of Software Evolution, IEEE, 2003, pp. 65-70

[Wa90] Wallmüller, E.: Software-Qualitätssicherung in der Praxis. Hanser, 1990.

[WDW08] Wagner, S.; Deissenboeck, F.; Winter, S.: Managing Quality Requirements Using Activity-Based Quality Models. In: Proceedings of the 6th International Workshop on Software Quality (WoSQ'08), ACM, 2008, S. 29-34.

[Yu95] Yu, E.: Modelling Strategic Relationships for Process Reengineering, Ph.D. thesis, University of Toronto, Ontario, Canada, 1995.