Maintainability is a Versatile Quality Attribute

Jens Knodel, Matthias Naab
Fraunhofer IESE
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{jens.knodel, matthias.naab}@iese.fraunhofer.de

***Abstract*— Software architecture evaluation has been widely accepted as a powerful means to mitigate risks in the design and evolution of software systems. To date we have conducted more than 75 architecture evaluation projects with industrial customers in the past decade. One recurring lesson learned that we experienced across many architecture evaluation projects is that maintainability indeed is a versatile quality attribute and its evaluation requires a mix of quantitative and qualitative checks.**

Keywords—software architecture, architecture evaluation, maintainability, reconstruction, reverse engineering, experience report

I. INTRODUCTION

There is no doubt about it: the quality attribute maintainability is crucial for the long-term success of a software system. The evaluation of maintainability consequently plays an important role in software evaluations. It contributes to answering questions like "Is our system a solid basis for the future?" or "Can we make the upcoming changes within a reasonable amount of time and budget?"

Evaluating maintainability can mean to check for different aspects: (1) it can be checked how adequate and maintainable an architecture and the underlying software system is in terms of supporting certain anticipated changes, (2) it can be checked how large the impact of a potential change request is and thus enable effort estimations, (3) it can be checked to which extent rules of good design and coding are adhered to, and (4) it can be checked how readable and understandable (and thus maintainable) the source code is. While the first two address specific properties of the product, the latter two address mental capabilities of software engineers.

The ISO 25010 definition of the quality attribute maintainability reflects this differentiation in its sub-qualities:

- *Modifiability / changeability* are aspects that mainly determine the overall change effort by the degree to which a concrete change is distributed over the whole system (from local to nearly global). This is mainly determined by major architecture decisions. Whether a change can be performed with low effort (meaning the system is maintainable) can be strongly influenced by architecture decisions, which cannot be measured locally in the code. For example, the requirement to replace the UI framework (e.g., because it is no longer supported by the vendor) might spread out over large parts of a system if there is no clear separation of the UI part and maybe even the concrete technology.
- *Readability / analyzability* are aspects that are mainly aiming at the understanding of the developers and strongly depend on the code quality (of course not exclusively; the overall architecture also has some impact here). That is, when developers have to change a certain part of the code, they first have to understand it. Good code quality obviously supports this. There has been a lot of research and numerous approaches exist regarding how to measure what good readability of code means. What is interesting now is that this type of code quality does not depend on the concrete product under development. Rather, it depends on the mental capabilities of the available developers. For example, if methods are too long or the degree of nesting is too high, they are hard to read, or cyclic dependencies are hard to understand. Finding good rules and thresholds thus has to be calibrated in empirical studies observing how well developers can work given certain code characteristics.

II. EVALUATING MAINTAINABILITY

Maintainability is a quality attribute with many indirections. Most of the time, it is not directly visible, in particular not for the end user, often not for the customer, and often not even for the developer. It is very common that there is not so much focus put on maintainability during initial system development (where time to market often predominates). The lack of maintainability and its consequences are perceived in later stages of system evolution and maintenance. Even then, the perception is mainly indirect, visible only in the high cost of changes. Another reason that makes maintainability harder to handle is that it is mostly difficult to precisely formulate maintainability requirements. Anticipated changes can be stated, but often it is pretty open how a system will evolve. In terms of readability, requirements are rather stated in terms of coding conventions than as real requirements.

A. Measuring Maintainability Quantitatively

When it comes down to measuring maintainability, this is not an easy task. In practice, the simple solution is often to buy a tool that measures code metrics and also outputs results on maintainability. These quantitative results can, of course, be valuable. However, they are only part of the answer. They are that part of the answer that deals with the readability / analyzability of the source code. The good thing is that it is easy to codify such rules and quantitative metrics and measure them with standard tools. This is often done in practice.

What is missing are considerations of major architectural decisions and concrete change scenarios of the software system. However, measuring this part of maintainability is not so easy for several reasons:

- Measurement needs concrete (anticipated) change requests as a baseline and often change requests that may occur further along in the future are not known yet.
- Measurement is not possible in absolute terms, but rather requires the usage of architecture evaluation techniques, which produce only qualitative results.
- Measurement is only possible manually with the help of experts; tool support is quite limited as the evaluation is individual for each specific product. Thus, this type of measurement is often neglected in practice and, as a consequence, maintainability is not measured sufficiently.

B. *Checking Adequacy of Solution Concepts for Maintainability Qualitatively*

There is no good or bad architecture – an architecture (or rather the solutions concepts defined by the architecture) always has to be adequate to satisfy requirements of the system at hand. More clarification is needed regarding what talking about the adequacy of "an architecture" means: An architecture is not a monolithic thing: It consists of many architecture decisions that together form the architecture. In the SAC, architecture drivers and architecture decisions are correlated. An architecture decision can support an architecture driver; it can adversely impact the driver; or it can be unrelated. Whenever it is not possible to observe properties in the running system or in local parts of the implementation, architecture becomes the means to provide the right abstractions for evaluating system properties.

Evaluating for maintainability benefits from a sound set of architecture drivers as input. The architecture drivers (in case of maintainability this means potential, possible or concrete change requests) are then evaluated qualitatively, the findings can be aggregated into an overall result. The checking of the adequacy works across "two worlds": requirements in the problem space and architecture in the solution space. There is no natural traceability relation between requirements and architecture. Rather, architectural decisions are creative solutions, which are often based on best practices and experiences, but sometimes require completely new approaches. This has an impact on the solution adequacy check: It offers limited opportunities for direct tool-supported analyses and is rather an expert-based activity.

The goal is to get the confidence that the solutions are adequate to be prepared towards change. As architecture is always an abstraction of the underlying software system, it typically does not allow for ultra-precise results. Thus, it should be made clear throughout an architecture evaluation which level of confidence needs to be achieved and what this means in terms of investment into evaluation activities.

Another form of qualitative evaluation works without concrete architecture drivers checking for the usage of architectural best practices like patterns or the SOLID principles. While these best practices can greatly support anticipated changes, they still can provide large benefits for unknown changes. However, the confidence achieved is lower.

C. *Interpretation of Quantitative and Qualitative Findings*

The interpretation of the findings is crucial in order to benefit from the overall evaluation results. The evaluation reveals typically positive and negative findings about both the code quality and the adequacy of the solution concepts. The interpretation of the findings is context-dependent, based on the underlying evaluation question, the software system under evaluation, and the nature of a finding. In particular in case of negative findings, it is the starting point towards deriving improvement actions.

The nature of findings is characterized by the cause of the finding. Causes can either be based on product properties such as the inherent flaws or weaknesses of the software system or technology and their misusage, or on human capabilities and human limitations in terms of coping with complexity (i.e., comprehensibility) and dealing with continuous change in the evolution of the software system (which is the inevitable characteristic of any successful system).

Typically, findings caused by technologies in use and limitations of human capabilities are more general. They serve to detect risks with respect to best practices, guidelines, misusage, or known pitfalls in the technologies. In practice, it is much easier to aim for such causes because tools can be bought that come, for instance, with predefined rules or standard thresholds and corridors for metrics. They are easy to apply and make people confident that they are doing the right thing to mitigate risks. The downside is that such general-purpose means often do not fit to the evaluation questions driving the architecture evaluation, but this is not discovered.

III. CONCLUSIONS

What is common sense for testing (no one would just run technology-related test cases without testing the product-specific logic of the software system) is not the case for architecture evaluations and source code quality checks. Here, many development organizations and tool vendors claim to be product-specific with a general-purpose technique realized in a quality assurance tool that can be downloaded from the Internet. However, these general-purpose techniques can merely lead to confidence regarding the avoidance of risks with respect to the technology or the limitations of human capabilities. To reliably make statements about the maintainability of a software system, both aspects, quantitative measurement of the source and qualitative evaluation of the adequacy of the solution concept are essential. Our lessons learned for maintainability are partly transferrable to other quality attributes such as performance, security, or reliability exhibit similar properties, meaning that code quality and architectural solution adequacy are both crucial for fulfilling the requirements.

Another characteristic of maintainability (which has increasing importance due to demanding requirements regarding time-to-market) is beyond the paper's scope: Changing a software system more and more needs a strong and dedicated integration, testing, and delivery pipeline which allows bringing the changed system in production with a high confidence in the resulting quality. This covers process, tooling, and architectural aspects that need to be well aligned to achieve the maintenance and release goals.

REFERENCES

[1] J. Knodel, M. Naab: „Pragmatic Evaluation of Software Architectures", Springer, ISBN 978-3-319-34176-7, 2016.