Evaluating Maintainability in Software Architecture

Gautam Kumar Instructor: Prof. Mark Kochanski

February 28, 2016

1 Introduction

A typical business software project spends 80% of its time in maintainance [10]. So its no wonder that maintainability is considered one of the most important quality attributes. Despite this assertion, evaluating and improving maintainability is largely procrastinated upon until the task becomes a signifanct burden on the budget. One of the ways to solve this problem would be to evaluate maintainability early on in the software development lifecycle and thus introduce a culture where engineers and architects think about maintainability as an important factor just like useability and performance. So with the goal of introducing a culture of maintainability early in mind, this white paper aims to help architects understand some the techniques of evaluating software architectures for maintainability.

2 Software Maintenance and Maintainability

Software maintenance as defined by the International Organization for Standardization is the process of modifying a software product after delivery to correct faults or to improve any quality attributes [1]

Maintainability is a quality attribute that describes the ease of introducing modifications into the software systems. Maintainability has four sub-characteristics namely analysability, changeability, stability and testability[1].

2.1 Types of Maintenance

ISO defines four types of software maintenance, Corrective, Preventive, Adaptive and Perfective.

Corrective maintenance refers to modifications done fix actual errors in the product while preventive maintenance is performed to introduce changes that can potentially prevent problems during the normal operation of the software product.

Adaptive and perfective maintenance are modifications introduced to enhance the core functionality of the software by improving one or more of its quality attributes.

3 Quantifying Maintainability

Understanding maintainability as a quality attribute is useful in learning the benefits of its presence in a product but to derive value from the process of evaluating software maintainability we require a method of measuring maintainability as a metric.

3.1 UML based metrics

One way to measure maintainability is to analyse the class diagram. This is especially useful in projects which use an object oriented programming language and have an existing UML diagram describing the software structure.

Genero et al [4] [3] describe an elegant way of predicting the maintainability of software early in the design phase by measuring the size and structural complexity of the software project.

UML based metrics cannot independently describe the maintainability of a system because they suffer from the same basic flaw that UML adoption faces. The fact that UML isn't as widely adopted in the software industry is the primary limiting factor. Though UML is extremely well defined and documented its complexity and the learning curve needed to effectively adopt UML in an organisation setting serves as a serious impediment to its adoption which in-turn reduces the value of using an UML based metric in measuring the maintainability of a software architecture.

3.2 Coupling as a metric

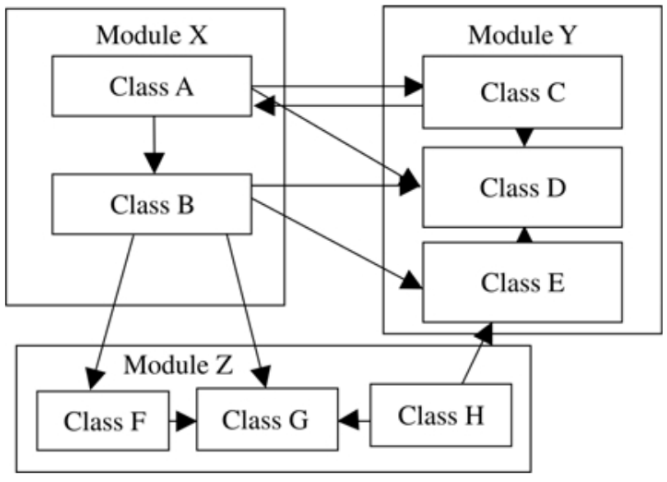

Figure 1: An example of architectural design. Arrows indicate coupling between the classes. Source:[7]

Coupling is the degree of interdependence between software modules [1]. Low coupling is often associated with high readability and maintainability because data and control flow often tends to be well structured in systems with low coupling.

Lindvall et al define[7] metrics to evaluate the maintainability of a software architecture based on the degree of coupling between modules in the software system. The metrics defined by the authors are

1. CBM coupling-between-modules
2. CBMC coupling-between-module-classes
3. CIM coupling inside a module

"CBM" is defined as the number of non-directional, distinct, inter-module references. Similarly "CBMC" is the number of non-directional, distinct, inter-module, class-to-class references for a module. Figure 1 exhibits an example of how "CBM" can calculated from an architecture diagram.

The goal when constructing a software architecture is to reduce Inter module dependencies (CBM and CBMC) at the expense of increasing intra module dependencies. Conversely when comparing architectures the one with lower CBM and CBMC scores are better.

3.3 Related quality attributes

Maintainability has a positive relationship with availability, flexibility, reliability, testability and a negative relationship with efficiency [6]. This means that an increase in flexibility, for example, could potentially improve the maintainability of the software system. An increase in efficiency could potentially hamper maintainability.

Understanding these trade off is important because indirect measurements as a factor of one of the aforementioned quality attributes can play a critical role when direct measurements of maintainability aren't available.

4 Optimal Maintainability

The optimal maintainability of a software project is usually not clearly defined. Architects evaluating various architectures try to reach an arbitrary value for maintainability based on experience and gut-feel. This can be an issue because a simple measure of maintainability does not offer much value interms of end goals. Such lack of clarity in a resource constrained environment would force software architects to focus their immidiate efforts into other clearly defined quality attributes, thus disregarding maintainability until its too late.

Bosch et al [2] propose a technique for evaluating a software architecture with respect to optimal maintainability. The authors classify maintainance activities into three categories, Adding new components, Adding plugins to existing components and changing existing component code.

The authors mention that earlier studies [5] [9] have shown that productivity of developing new components is much higher than building plugins and modifying existing component code is shown to be significantly slower than building new components or plugins. So the goal of the first exercise is to build a maintainance profile with a set of possible change scenarios classified as adding new components, plugins or changing existing components.

The maintainance profile can now be used to compute the maintainance effort required for implementation. An impact analysis, which accounts for the number of components added, changed and also the number of plugins created, is also performed for the maintainance profile.

Using the data from impact analysis and maintainance effort we can compute the optimal maintaince effort needed for the change scenario using the aforementioned assumption that

an ideal software architecture would allow for change scenarios to be implemented purely by adding new components or in the case of small changes adding new plugins.

5 Other techniques for evaluating Maintainability

- Maintaining Maintainability [11]
  In this paper the authors attempt to quantify the effects of people, organisation and the maintenance process as a metric to evaluate how these external factors affect maintainability in a software system.

- Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability [8]
  This paper is a meta analysis of various methods of software architecture evaluation for to quantify quality attributes such as Performance, Testability, Maintainability and portability.

6 Conclusion

Quality attributes such as performance and maintainability are usually not clearly defined, if at all, in the requirements specification because stakeholders do not generally understand their necessity or value. It is our job as software architects to evaluate and include maintainability as part of the architecture because maintainability offers one of the highest value in the longer run of any project. In this white paper we've highlighted some techniques of evaluating the maintainability of a software architecture before implementation and thus benchmarking the expectations for the production system.

Biblography

[1] , I. International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering #2013; Software Life Cycle Processes #2013; Maintenance. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)* (2006), 0_1–46.

[2] Bosch, J., and Bengtsson, P. Assessing optimal software architecture maintainability. In *Fifth European Conference on Software Maintenance and Reengineering, 2001* (2001), pp. 168–175.

[3] Genero, M., Manso, E., Visaggio, A., Canfora, G., and Piattini, M. Building measure-based prediction models for UML class diagram maintainability. *Empir Software Eng 12*, 5 (Mar. 2007), 517–549.

[4] Genero, M., Piattini, M., Manso, E., and Cantone, G. Building UML class diagram maintainability prediction models based on early metrics. In *Software Metrics Symposium, 2003. Proceedings. Ninth International* (Sept. 2003), pp. 263–275.

[5] Henry, J. E., and Cain, J. P. A quantitative comparison of perfective and corrective software maintenance. *Journal of Software Maintenance: Research and Practice 9*, 5 (1997), 281–297.

[6] Karl, W. Software requirements. *Edition-2, Microsoft* (2003).

[7] Lindvall, M., Tvedt, R. T., and Costa, P. An Empirically-Based Process for Software Architecture Evaluation. *Empirical Software Engineering 8*, 1 (Mar. 2003), 83–108.

[8] Mattsson, M., Grahn, H. a., and M\a artensson, F. Software architecture evaluation methods for performance, maintainability, testability, and portability. In *Second International Conference on the Quality of Software Architectures* (2006), Citeseer.

[9] Maxwell, K. D., Van Wassenhove, L., and Dutta, S. Software development productivity of European space, military, and industrial applications. *Software Engineering, IEEE Transactions on 22*, 10 (1996), 706–718.

[10] Pigoski, T. M. *Practical software maintenance: best practices for managing your software investment*. John Wiley & Sons, Inc., 1996.

[11] Ramage, M., and Bennett, K. Maintaining maintainability. In *, International Conference on Software Maintenance, 1998. Proceedings* (Nov. 1998), pp. 275–281.