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 maintenance [Pigoski,]. 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 significant burden on the budget. One of the ways to solve this problem would be to evaluate maintainability early on in the software development life-cycle 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 [ISO,]

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[ISO,].

The paper [Heitlager et al.,] provides easy to understand definitions for analysability, changeability, stability and testability.

Analysability is a measure of the ease or difficulty in diagnosing the system for deficiencies or identifying the parts that need modification.

Changeability is a measure of the ease or difficulty in making adaptations to the system in a reasonable amount of time.

Stability is a measure of the ease or difficulty in maintaining the system in a consistent state during modifications.

Testability is a measure of the ease or difficulty in testing the system after modifications have been completed.

2.1 Types of Maintenance

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

Corrective maintenance refers to modifications done fix actual errors in the product. Corrective maintenance is usually to fix bugs that might have cropped up in previous versions of the software.

Preventive maintenance is performed to introduce changes that can potentially prevent problems during the normal operation of the software product. Examples of preventive maintenance would be introducing security checks such as Input sanitization to prevent SQL Injection or XSS attacks, implementing validations on user input to prevent user from accidentally moving the system to an unstable state.

Adaptive and Perfective maintenance are modifications introduced to enhance the core functionality of the software by improving one or more of its quality attributes. An example of adaptive maintenance would be introducing a change request from the customer on the way certain features work while perfective maintenance would be improving performance or user experience of existing features.

Emergency maintenance is any unscheduled modification of the system during emergency situations such as during an unexpected spike in load, a cyber attack or unplanned last minute changes before a presentation.

3 Evaluating 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 methods of measuring maintainability as a metric.

Predicting maintainability of a software system early in its development life-cycle is a difficult problem containing a lot of uncertainty. Consequently the effective early prediction of software system depends on using indicators available on early development documents such as UML diagrams, SAVN diagrams and the requirements specification documents.

Uncertainty associated with quantifying maintainability can greatly be reduced if a reasonable amount of effort is invested in building detailed architecture documentation using tools such as UML diagrams and applying structured processes such as SAAM (Software Architecture Analysis Method) and ATAM (Architecture Tradeoff Analysis Method) to evaluate the proposed architecture.

3.1 Quantifying Maintainability

3.1.1 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.

In [Genero et al.,] the authors 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.

Size metrics defined by [Genero et al.,] are Number of Classes (NC), Number of Attributes (NA), Number of Methods (NM). These metrics are usually the easiest to quantify. As a standalone metric, size is a poor indicator of maintainability because A small project can potentially be unmaintainable because of highly coupled design while a project with a large number of classes might be easily maintainable due clean interfaces and loose coupling. As a consequence size metrics are used in combination with other factors to come up with a maintainability score.

Structural complexity metrics as defined by [Genero et al.,] are Number of Associations (NASSOC), Number of Aggregations (NAGG), Number of Dependencies (NDEP), Number of Generalisations (NGEN), Number of Generalisation hierarchies (NGENH), Number on Generalisation hierarchies (NAGGH). Structural complexity metrics are relatively harder to measure when compared to size metrics but offer a much better understanding of the interdependencies between components on the system.

Conclusions from [Genero et al.,] are that Understandability/Analysablility and Modifiability/Changeability are affected by measures of NAGGH, NGENH and NA.

The primary issue with UML based metrics is that they 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 it in an organization 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.

3.1.2 Coupling as a metric

Coupling is the degree of interdependence between software modules [ISO,]. 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.

In [Lindvall et al.,] the authors define 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.

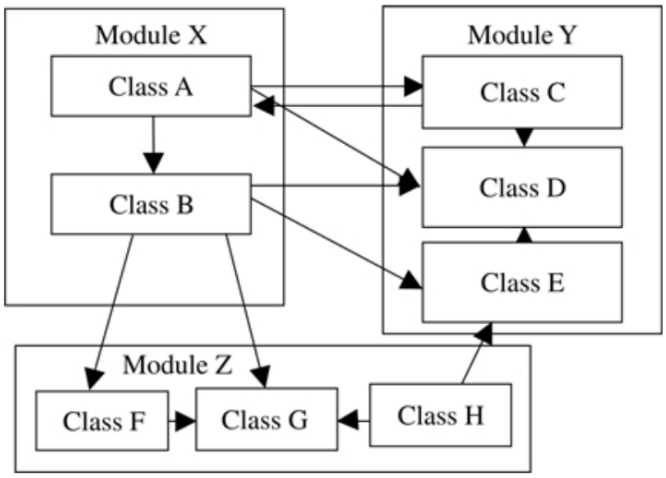

Figure 1: An example of architectural design. Arrows indicate coupling between the classes. Source:[Lindvall et al.,]

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.

An sample comparison of two designs from [Lindvall et al.,] is shown in figure 2 and 3. Each architecture has an expected coupling measure as elucidated by the table in figure 4. When EMS1 and EMS2 are evaluated the coupling measures obtained are shown in Figure 5 and 6. From these tables we can easily come to the conclusion that EMS2 has lower overall coupling except in the Mediator and Common modules which is intended thus we can conclude that EMS2 [3] is more maintainable than EMS1 [2].

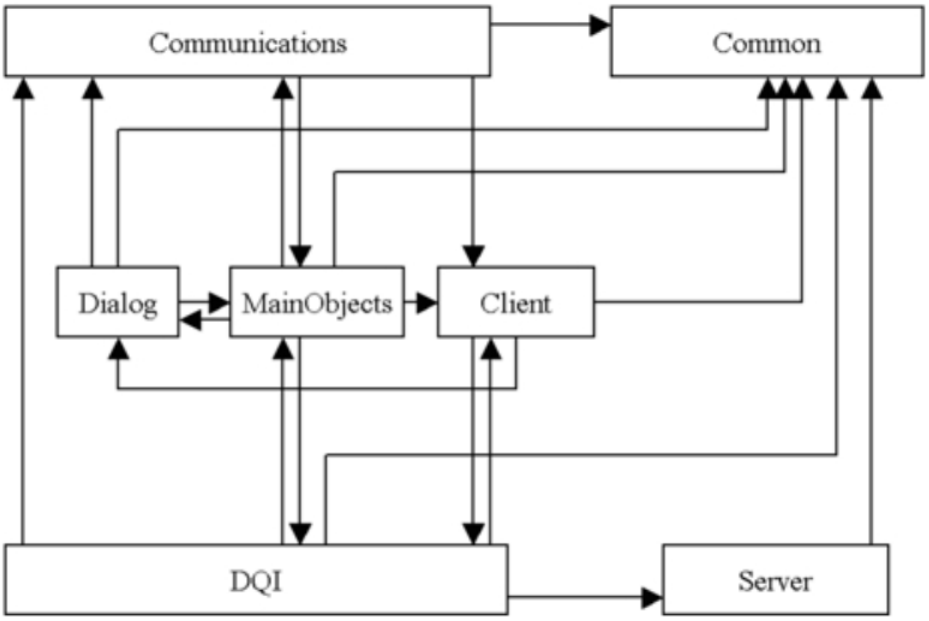

Figure 2: EMS1 [Lindvall et al.,]

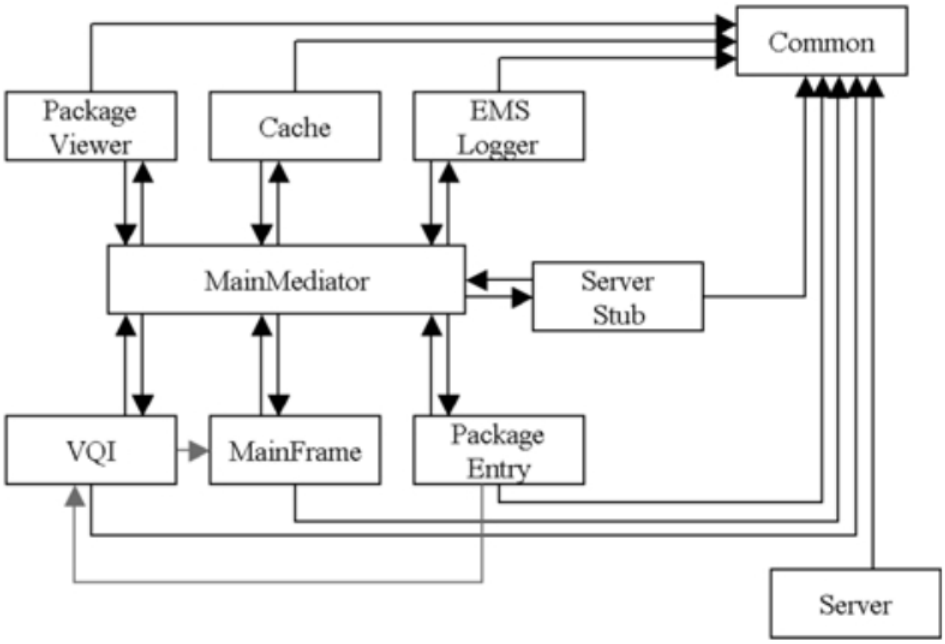

Figure 3: EMS2 [Lindvall et al.,]

Expectation	Explanation
CBM_{hist}(EMS2) better than CBM_{hist}(EMS1)	In EMS1, most modules will be highly coupled. In EMS2, most modules will have low CBM, less than or equal to two. The only exceptions will be the Mediator and Common modules.
$CBMC_{hist}$(EMS2) better than $CBMC_{hist}$(EMS1)	In EMS1, most modules will have high CBMC. In EMS2, most modules will have low CBMC, less than or equal to 10. The only exceptions will be the Mediator and Common modules.
CIM(EMS2) is nearly the same as CIM(EMS1)	In EMS1, modules will contain unrelated classes leading to lower CIM numbers. In EMS2, modules will contain related classes leading to higher, but not significantly higher CIM numbers.

Figure 4: Coupling expectations of Figure 2 and 3
source:[Lindvall et al.,]

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Client	**5**	1	10	9
Common	**6**	1	31	N/A
Communications	**5**	1.7	30	**22**
Dialog	4	1.3	18	**16**
DQI	**5**	4.5	39	**26**
Main Objects	**5**	1.6	41	**39**
Server	2	2	7	2

Figure 5: Coupling measures for EMS1 [2] [Lindvall et al.,]

3.1.3 Related quality attributes

Maintainability has a positive relationship with availability, flexibility, reliability, testability and a negative relationship with efficiency [Karl,]. 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.

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Cache	2	0	9	3
Common	**8**	2	85	
EMS Logger	2	1	4	3
Main Frame	*3*	3.9	14	7
Main Mediator	**8**	1	37	**25**
Package Entry	*3*	1	13	6
Package Viewer	2	4.3	27	4
Server	1	2.25	12	0
Server Stub	2	0	4	3
VQI	*4*	2.9	23	7

Figure 6: Coupling measures for EMS2 [3] [Lindvall et al.,]

Maintainability and Availability: Availability is a measure of the portion of time that the system is functional and working[microsoft,]. Consequently the higher the maintainability the better the availability of the system because issues are easier to identify and fix.

Maintainability and Reliability: Reliability is measured as the probability that a system will not fail and that it will perform its intended function for a specified time interval [microsoft,]. This attribute too has a logically positive relationship with Maintainability because reliability is often achieved through an iterative process and maintainability facilitates that iterative process.

Maintainability and Efficiency: Maintainability can potentially have a negative impact on performance / efficiency or vice versa because highly efficient code might need to use some low level tricks which would dramatically reduce readability and thus have a negative impact on maintainability. Conversely a maintainable system might be too loosely coupled and offer multiple layers of abstraction which would cause a noticeable impact on performance or efficiency.

	Availability	Efficiency	Flexibility	Integrity	Interoperability	Maintainability
Availability						
Efficiency			-		-	-
Flexibility		-		-		+
Integrity		-			-	
Interoperability		-	+	-		
Maintainability	+	-	+			
Portability		-	+		+	-
Reliability	+	-	+			+
Reusability		-	+	-		
Robustness	+	-				
Testability	+	-	+			+
Usability		-				

Figure 7: Maintianability and related quality attributes Source:[Morgan,]

These trade offs are explained further in Microsoft's article on the topic [Morgan,] where the author describes the relation between various quality attributes using an elegant chart in figure 7. 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.

3.2 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 in terms of end goals. Such lack of clarity in a resource constrained environment would force software architects to focus their immediate efforts into other clearly defined quality attributes, thus disregarding maintainability until its too late.

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

The authors mention that earlier studies [Henry and Cain,] [Maxwell et al.,] have shown that productivity of developing new components is much higher than building plug-ins and modifying existing component code is shown to be significantly slower than building new components or plug-ins. So the goal of the first exercise is to build a maintenance profile with a set of possible change scenarios classified as adding new components, plug-ins or changing existing components.

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

Using the data from impact analysis and maintenance effort we can compute the optimal maintenance 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 plug-ins.

3.3 Other techniques for evaluating Maintainability

- Maintaining Maintainability [Ramage and Bennett,]
  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 [Mattsson et al.,]
  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.

4 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

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

[Genero et al.,] Genero, M., Manso, E., Visaggio, A., Canfora, G., and Piattini, M. Building measure-based prediction models for UML class diagram maintainability. 12(5):517–549.

[Heitlager et al.,] Heitlager, I., Kuipers, T., and Visser, J. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39.

[Henry and Cain,] Henry, J. E. and Cain, J. P. A quantitative comparison of perfective and corrective software maintenance. 9(5):281–297.

[ISO,] ISO, I. International standard - ISO/IEC 14764 IEEE std 14764-2006 software engineering #2013; software life cycle processes #2013; maintenance. pages 0_1–46.

[Karl,] Karl, W. Software requirements.

[Lindvall et al.,] Lindvall, M., Tvedt, R. T., and Costa, P. An empirically-based process for software architecture evaluation. 8(1):83–108.

[Mattsson et al.,] Mattsson, M., Grahn, H. a., and M{\textbackslash}a artensson, F. Software architecture evaluation methods for performance, maintainability, testability, and portability. In *Second International Conference on the Quality of Software Architectures*. Citeseer.

[Maxwell et al.,] Maxwell, K. D., Van Wassenhove, L., and Dutta, S. Software development productivity of european space, military, and industrial applications. 22(10):706–718.

[microsoft,] microsoft. Quality attributes. https://msdn.microsoft.com/en-us/library/ee658094.aspx.

[Morgan,] Morgan, G. Implementing system-quality attributes.

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

[Ramage and Bennett,] Ramage, M. and Bennett, K. Maintaining maintainability. In *, International Conference on Software Maintenance, 1998. Proceedings*, pages 275–281.