Service oriented architecture evaluation based on maintainability index

Edgaras Norvaiša
Kaunas University of Technology
Faculty of software engineering
Kaunas, Lithuania
e-mail: edgaras.norvaisa@ktu.edu

Šarūnas Packevičius
Kaunas University of Technology
Faculty of software engineering
Kaunas, Lithuania
e-mail: sarunas.packevicius@ktu.edu

***Abstract*—Software architecture is meant to define system structure which is a core for product development. Defying system structure is first and most important step in software development. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application. Most efficient way to be able to control system characteristics is to use certain architectural models like SOA or any other. In this paper focus object will be how maintainability is effected for SOA based applications.**

Keywords—SOA; metrics; maintainability

I. INTRODUCTION

Decision to analyze SOA maintainability was made because most of the enterprise projects chose to use SOA as a base for architecture but this does not mean that it guaranties good application characteristics. This research is focused to analyze and define how maintainability index is changing dependently on using SOA structure by comparing different system layouts. Main goal of experiment was to find out how maintainability index changes for SOA based application and what is effecting these changes. To initiate experiment analysis of SOA and software metrics was made. Explaining principles of SOA architecture and metrics that are used to measure software maintainability. Experiment part describes several approaches how SOA can be structured and how that structure impacts maintainability for whole application. The result will allow determine how maintainability is effected by using SOA principles and what architectural approach should be chosen to rise system maintainability.

II. DESCRIPTION OF SERVICE ORIENTED ARCHITECTURE

A. Definition of SOA

Service oriented architecture as Mamoun A. Hirzalla explains "SOA is emerging as a promising development paradigm, which is based on encapsulating application logic within independent, loosely-coupled stateless services, that interact via messages using standard communication protocols and can be orchestrated using business process languages, The notion of a service is similar to that of a component, in that services, much like components, are independent building blocks that collectively represent an application."[8]. This statement describes that SOA focuses on having loosely-coupled services that are like independent building blocks used for building application. The benefit of SOA is that it provides loosely coupling where in a result we have flexible, scalable and reusable application.

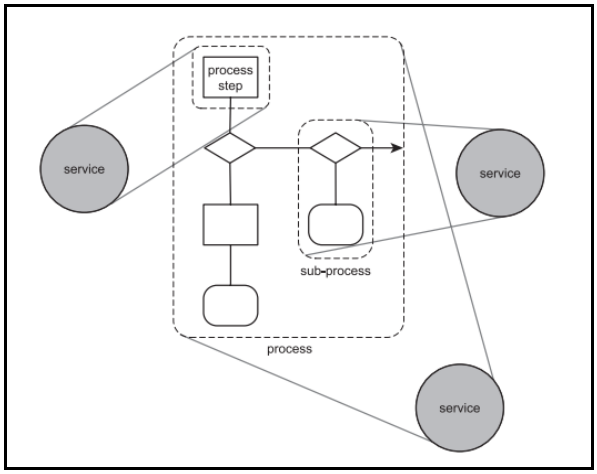

Figure 1. Distributed application services.

Figure 1 illustrates how services can look like and how they can interact with each other. It shows that a set of services can be a part of a bigger process that can be used to describe a larger business cases or algorithm. To process a certain business case, we call a service which is responsible for it. Service can be built from other smaller services which implement smaller parts of a main service. To communicate with services a communication layer is needed which would provide interface and description of network services. Services can be accessed directly an invoking client or through service broker (ESB) which looks up the address of required services through a registry component, retrieves the Web Service Definition Language (WSDL) file, and then binds to that service during the invocation process. ESB responsible for routing and translating requests and responses between requestors and service providers (e.g. figure 2). In the context of Service Oriented Architecture, Web Services are used to facilitate communication between service providers and service consumers. Web based applications are communicating using the concepts as XML, SOAP, REST, WSDL and UDDI. [5]

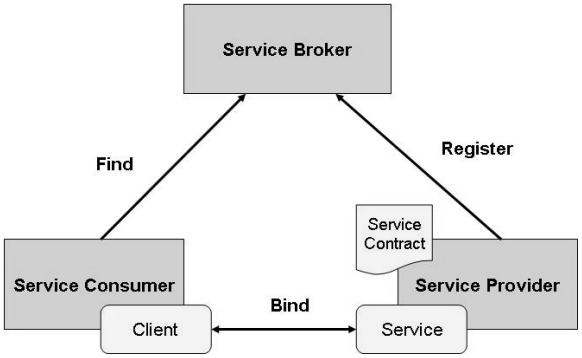

Figure 2. SOA design.

As mentioned before SOA is paradigm, which is based on application logic encapsulation. In such case, there is a need to have a structure for encapsulated logic – services [1].

B. *N-tier architecture*

In Christoph Hartwich article about difficulties in building n-tier enterprise application he describes how application can be divided in to separate tiers. "A tier is a layer that corresponds to a process or a collection of processes. A tier contains all artifacts of a software system that can be associated with the tier's process(es)." [2]. N-tier approach divides application architecture in to separate layers which are loosely coupled. Application can have multiple tiers but in most cases, there are main 3 tiers. Each tier has its own responsibility and it's only accessible for neighbors and greater tier. Top tier is a presentation tier which provides an interface for clients to business logic. Second tier is a business logic where system logic is predefined in to services. Business tier can be accessed only by presentation tier where business tier can access data tier. Data tier is responsible for granting access to repositories. Data tier can be accessed only by business tier services (e.g. figure 3).

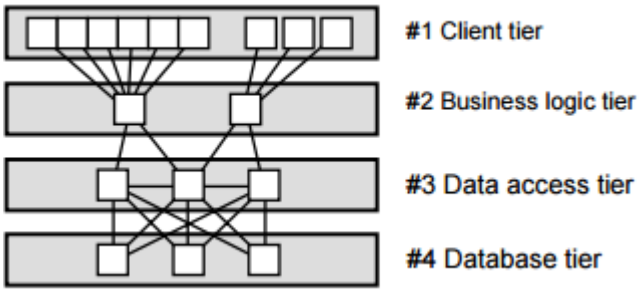

Figure 3. N-tier structure.

To summarize SOA is an enterprise architectural style rather than an application architectural style. SOA doesn't focus on individual application architecture where n-tier does it. In the end, we have a set of distributed services across the application which are published by service broker and can be accessed by other consumers (e.g. figure 4).

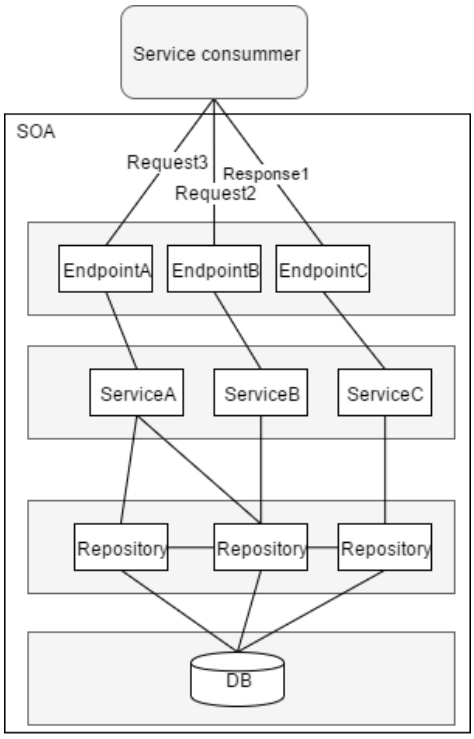

Figure 4. SOA based design.

III. SOFTWARE METRICS

To define software quality and complexity software metrics are used. Software metrics are widely used among software developers and clients who makes requirements for their software quality. There are multiple methods and algorithms how to calculate software metrics but these days in most case Halstead techniques are used. In this topic, we will define few of the metrics that was used to calculate maintainability index in experiment [6].

A. *Lines of code (LOC) metric*

One of the metrics is line of code (LOC) which is used to determine size of software. Using this metric, it is possible to evaluate size of a software but to do it correctly a prime calculation object need to be defined. It can be done in multiple ways [7]:

- Calculate only executable lines
- Calculate executable lines and assignments of values
- Calculate executable lines, assignments and comments
- Calculate lines in input screen

Because of multiple approaches of how code lines can be calculated LOC is not very inaccurate metric although it's used in maintainability metric calculation.

B. *McCabe's cyclomatic complexity metric*

Second important metric in evaluating software is McCabe's cyclomatic complexity. It is one of the most used software metrics. This metric was invented by Thomas J. McCabe 1976 year. This is one of the most accurate software metrics that can be calculated and is widely used in software which calculating metrics of other software programs.

Cyclomatic complexity (CC) is classical graph theory cyclomatic number, indicating the number of regions in a graph. As applied to software, it is the number of linearly independent paths that comprise the program. As so it can be used to indicate effort required to do testing. This metric was design to indicate a program's testability and maintainability [7]. The general formula to compute it is:

$$M = V(G) = E - N + 2P \quad (1)$$

- V(G) – Cyclomatic number of G
- E – Number of edges
- N – Number of nodes
- P – Number of unconnected parts of the graph

In functional programing calculation of cyclomatic complexity is easy because each operation is executed after each other. In OO (object oriented) case it is much complicated because operations are divided in functions, classes and objects. Best approach to calculate CC for OO is to calculate it per functions. By McCabe CC should not be bigger than 10 so if function or a method has bigger CC than 10 code needs to be simplified or divided. Using this metric for software evaluation it allows to optimize a whole size and complexity of application as well as increase maintainability [7].

C. *Halstead metrics*

Last metric which is used in MI calculation is Halstead volume. Halstead software science measurements where introduced in 1977 to estimate the program difficulty and other features like development effort and project number of faults. A computer program, according to software science, is a collection of tokens that can be classified as either operators or operands [7]. The measures of Halstead are based on

- n1 = number of unique or distinct operators
- n2 = number of unique or distinct operands
- N1 = total usage of all the operators
- N2 = total usage of all the operands

Based on these primitive measures Halstead developed a system of equations expressing the total vocabulary. Following metrics [7]:

$$N = N1 + N2 \quad (2)$$

$$n = n1 + n2 \quad (3)$$

$$V = N \log_2(n) \quad (4)$$

$$D = \frac{n1}{2} * \frac{N2}{n2} \quad (5)$$

$$E = D * V \quad (6)$$

- N – program length
- n – program vocabulary
- V – volume
- D – difficulty
- E - effort

D. *Maintainability index*

Maintainability index is a composite metric that incorporates several traditional source code metrics into a single number that indicates relative maintainability. As Oman and Hagemeister explains MI is comprised of weighted Halstead metrics (effort or volume), McCabe's Cyclomatic Complexity, lines of code (LOC), and number of comments [3] [4]. The original formula to compute MI is:

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) \quad (7)$$

- MI – maintainability index
- HV –Halstead volume per module
- CC – cyclomatic complexity per module
- LOC – lines of code

Having full understanding of how software can be evaluated and measured this knowledge was used to initiate experiment for evaluating SOA based software maintainability index. In next topics experiment methodology will be defined.

IV. METHODOLOGY OF THE RESEARCH

To evaluate maintainability index for SOA principles based application experimental program was created. In experimental part maintainability will be measured using different application structure layouts. System structure in most of the cases will be service oriented so there will be a set of services that will implement certain logic of one of the main service. Main service will consume other services in different ways – straight calls to service or will use other services to call other service operation. Basic layout of all structures will be based on n-tier principles so there will be free layers – presentation tier which will only define main service interface and provide ability to call it from other clients. Business tier will be responsible for all the services that will be used to solve a main service operation, as well as main service will be in this tier. Repository tier will be responsible for data access and all upper tier services will be able to consume these repositories.

For calculating metrics in experiment integrated Visual Studio tools were used because experiment is executed on .NET environment using C# programing language. Formula used for measuring maintainability in Visual Studio is the same. The new definition merely transforms the index to a number between 0 and 100. Furthermore, Visual Studio provides an interpretation:

- MI >= 20 high maintainability
- 10 <= MI < 20 moderate maintainability
- MI < 10 low maintainability

In experiment four different structures will be measured. First structure in figure 5 contains of only main service which calls data insert operations in ten repositories. Where:

- E – end point for client to execute main service
- MS – main service which handles main application logic operation
- SX – other services that are consumed by main service
- RX – repositories that provides access to data insert operations.

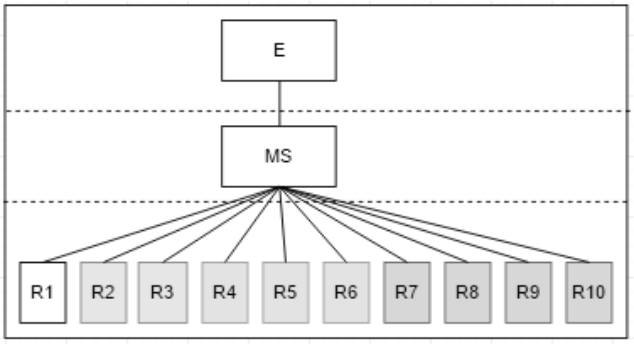

Figure 5. Experiment scenario 1.

Second structure consists of 4 services where main service is calling S1 service operation which is calling repositories R2, R3. S1 operation is calling S2 operation which works in same principle as S1 only number of consumed repositories is greater. Rest application structures are displayed in 6-8 figures. Main difference in structure is that the main service operation is distributed across the multiple services. Each structure only has a different operation call flow. As for example man service in figure 7 is calling distributed S1, S2 and S3 services to complete main MS operation.

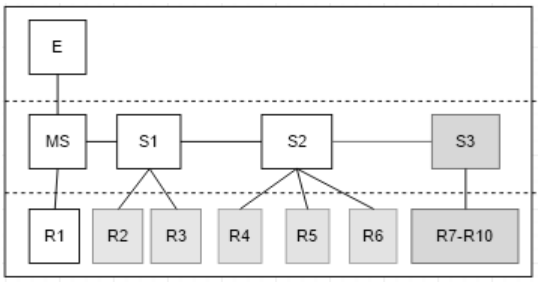

Figure 6. Experiment scenario 2.

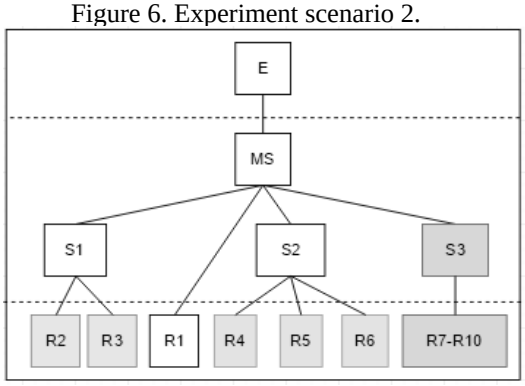

Figure 7. Scenario 3.

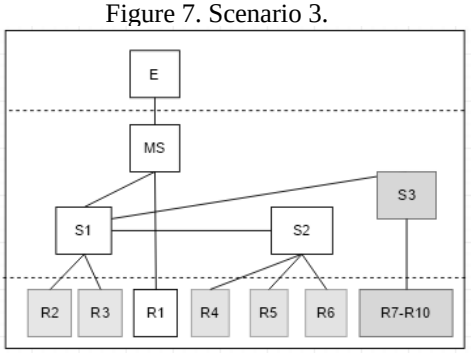

Figure 8. Scenario 4.

V. Experiment results

Experiment results of scenario 1 (Figure 5) are represented in table 1. Each row represents experiment step where we add a new service or repository to application structure. At the first scenario, the main service is not calling any other service it only calls repositories that allows completing main service logic-operation. This kind of structure is illustrating a default cases in applications where we don't use distributed services (non SOA approach). Service column in a table 1 defines how many services are consumed by the man service. Same approach goes for repositories column. Metrics column represents all the metrics like maintainability index, cyclomatic complexity and code coupling which were calculated by Visual Studio integrated tools. After executing experiment for first scenario we can see that MI is decreasing. This happens because each time we add a new repository to main service a complexity of application is increasing.

TABLE I. Scenario 1 results

Services	Repositories	Metrics		
		MI	***CC***	***Coupling***
0	1	93	3	2
0	2	92	4	3
0	3	89	5	4
0	4	88	6	5
0	5	86	7	6
0	6	86	8	7
0	7	85	9	8
0	8	84	10	9
0	9	84	11	10
0	10	84	12	11

Second scenario (Figure 6) results are represented in table 2. In this scenario, we define that main service on each experiment step starts to consume a new service which is providing functionality that resolves a small logical case of the main service operation. Also, after each experiment step we connect a new service which has more repositories in use then the previously added service. Compare to table 1 we can see that maintainability index is decreasing much slower. Despite that fact that cyclomatic complexity and code coupling is increasing much faster. Results of how maintainability index decreases are displayed in figure 9.

TABLE II. Scenario 2 results

Services	Repositories	Metrics		
		MI	***CC***	***Coupling***
0	1	93	3	2
1	2	92	7	1
1	3	92	8	5
2	4	91	12	7
2	5	91	13	8

Services	Repositories	Metrics		
		MI	*CC*	*Coupling*
2	6	90	14	9
3	7	90	18	11
3	8	90	19	12
3	9	90	20	13
3	10	89	21	14

Comparing the first and the second scenario results difference is noticeable on how fast MI is decreasing. Difference can be seen in figure 9. The reason why it happens is related to Halstead Volume metric which measures volume of algorithm. In this experiment algorithm, basically is the operations that main service is executing. On second scenario, main service operation logic is distributed across the other services so average Halstead Volume is smaller than it is on a first scenario. This is also can be seen on scenarios figures 5 and 6 of how many relations MS has to other objects. On figure 5 main service operation consists of several operations that are calling 10 repositories. On the other case in figure 6 we see that main service is only calling 1 repository and 1 service so in total main service operation consists of 2 calls. In scenario 2 rest main service operation logic is distributed in S1, S2 and S3 services but all these services consist of few operations that calls repositories and other service. In the end at scenario 2 we have average Halstead Volume metric which is smaller than in a first scenario.

Remaining scenarios showed in figures 7 and 8 results are the same as we had in scenario 2. Maintainability index changes in a same number only other metrics like CC is different. Reason again is related to Halstead Volume metric where we have a different value across whole modules-services but in the end maintainability index is calculated using average Halstead Volume value.

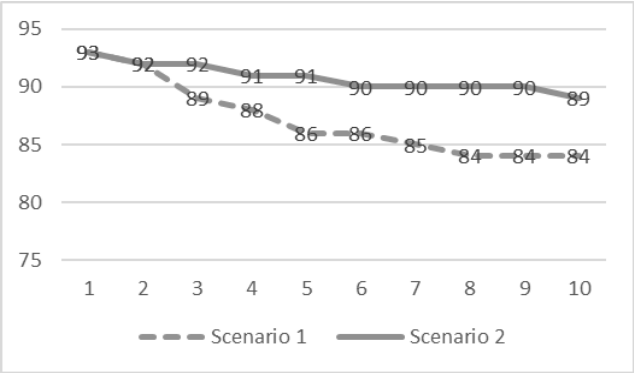

Figure 9. Scenario 1 and 2 chart of MI metrics per experimental step.

VI. Correlation between metrics

After experiment correlations where calculated for different types of metrics. For all calculations maintainability index was used as a prime metric and secondary metric was used from MI formula. Table 3 represents calculations results where first and second values (CC, Code coupling) has strong negative values. This means that if one of these metrics are increasing maintainability index is decreasing. Third metric – Halstead Volume has strong and positive correlation between MI. This means that when HV is increasing MI is increasing to. This positive correlation between MI and HV was noticed during experiment part when code metrics where measured for different system layouts in topic V.

TABLE III. Correlations between metrics

Nr.	Correlations with MI and metric	
	Metric	*r*
1	Cyclomatic complexity (CC)	-0.96
2	Code coupling	-0.94
3	Halstead Volume (HV)	0.85

VII. Conclusions

1. Analysis part shows that SOA allows to have loosely coupled application components that can be distributed not even across application scope but also outside it.
2. Metrics analysis part shows that there are methods that allows evaluate software quality and complexity.
3. Experimental evaluation of different application layouts gave different maintainability index results. In case when we have no distributes services across application it becomes harder to maintain the application. In another case when we have distributed services (service oriented) maintainability of application is much easier. However, this conclusion is only applicable when we have a bigger scale application. This is one of the reason why SOA is used in enterprise.
4. Experimental part allows to make conclusion that only in situation when we distribute application logic Halstead Volume metric increases and in this way, it effects maintainability index to be increased.
5. Practical benefit of this research is that we can see that SOA approach is better for bigger scale application instead of small size projects. Also, research clearly shows that having big logical operations in one place makes system hardly maintainable.
6. For easy maintainable system, best architectural approach is to use n-tier by distributing system in to layers – vertically. Also, it is good to distribute system logic across the service – horizontally.
7. For SOA development recommendation is to keep services as much as possible isolated. This will reduce code coupling and increase code maintainability as well as reusability.

REFERENCES

[1] S. K. Simardeep Kaur, "A Review on Metrics in SOA," 2016. [Tinkle]. Available: http://www.ijarcce.com/upload/2016/july-16/IJARCCE%2081.pdf. [Accessed 20 02 2017].

[2] C. Hartwich, "Why It Is So Difficult to Build N-Tiered Enterprise." Freie Universitaet Berlin, Berlin, 2001.

[3] R. Land, "Measurements of Software Maintainability." Mälardalen University, Västerås, Sweden.

[4] K. D. Welker, "The Software Maintainability Index Revisited." Idaho National Engineering and Environmental Laboratory, 2001.

[5] T. Erl, "Service-Oriented Concepts, Technology, and Design." United States: R. R. Donnelley in, 2005.

[6] L. Westfall, "12 Steps to Useful Software Metrics," The Westfall Team, 2005.

[7] S. H. Kan, "Metrics and Models in software quality engineering." Canada: Person education inc., 2003.

[8] M. A. Hirzalla, "Service Oriented Architecture Metrics Suite for Assessing and Predicting Business Agility Results of SOA Solutions." Chicago, Illinois: College of Computing and Digital Media, 2012.