Predicting Web Service Maintainability via Object-Oriented Metrics: A Statistics-based Approach

José Luis Ordiales Coscia[2], Marco Crasso[1,2,3], Cristian Mateos[1,2,3], Alejandro Zunino[1,2,3], and Sanjay Misra[4]

[1] ISISTAN Research Institute.
[2] UNICEN University.
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).
[4] Department of Computer Engineering, Atilim University, Ankara, Turkey

Abstract. The Service-Oriented Computing paradigm enables the construction of distributed systems by assembling loosely coupled pieces of software called services, which have clear interfaces to their functionalities. Service interface descriptions have many aspects, such as complexity and quality, all of which can be measured. This paper presents empirical evidence showing that services interfaces maintainability can be predicted by applying traditional software metrics in service implementations. A total of 11 source code level metrics and 5 service interface metrics have been statistically correlated using 154 real world services.

Keywords: SERVICE-ORIENTED COMPUTING; WEB SERVICES; CODE-FIRST; WEB SERVICE MAINTAINABILITY; OBJECT-ORIENTED METRICS; WEB SERVICE MAINTAINABILITY PREDICTION.

1 Introduction

Service-Oriented Computing (SOC) is a paradigm that allows engineers to build new software by composing loosely coupled pieces of existing software called *services*. A distinguishing feature of SOC is that services may be provided by third-parties who only expose services interfaces to the outer world. By means of these interfaces, potential consumers can determine what a service does from a functional perspective and remotely invoke it from their new applications.

The advances in distributed system technologies have caused engineers to materialize SOC in environments with higher levels of distribution and heterogeneity. The availability of broadband and ubiquitous connections enable to reach the Internet from everywhere and at every time, creating a global scale marketplace of software services where providers offer their services interfaces and consumers may invoke them regardless geographical aspects, using the current Web infrastructure as the communication medium. Therefore, services are often implemented using standard Web-inspired languages and protocols and thus are called *Web Services*. Nowadays, Web Services are the technology commonly used when migrating legacy systems [1] to modern platforms or the technological protocol stack used for accessing information from smartphones [2].

Like any other software artifact, service interface descriptions have a size, complexity and quality, all of which can be measured [3]. In fact, previous research [4,3,5]

has emphasized on the importance of the non-functional concerns of services interfaces. Particularly, the work of [4] proposes a catalog of common bad practices found in services interfaces, which impact on the understandability and discoverability of the associated services. Understandability is the ability of a service interface description of being self-explanatory, which means a software engineer can reason about a service purpose just by looking at its interface description. Discoverability refers to the ability of a service of being easily retrieved, from a registry or repository, based on a partial description of its intended functionality such as a Google-like query. At the same time, in [3] the author describes a suite of metrics to assess the complexity and quality of services interfaces with respect to various aspects. Likewise, [5] proposes a suite comprising 4 metrics to assess service maintainability from service interface descriptions.

Methodologically, in practice service interfaces are not build by hand, but instead they are automatically generated by mapping programming languages constructors (and hence service implementations) onto service interface descriptions expressed in the Web Service Definition Language (WSDL). WSDL is an XML-based format for describing a service as a set of operations, which can be invoked via message exchange. In the Java arena, this mapping is usually achieved by tools such as Axis' Java2WSDL, Java2WS, EasyWSDL, and WSProvide. The weak point of this methodology is that engineers partially control WSDL specification and therefore resulting WSDL descriptions may suffer from understandability, discoverability, complexity, quality and maintainability problems as measured by the aforementioned metrics catalogs.

We set forth the hypothesis that service developers can indirectly reduce the negative impact of some of these WSDL-level metrics by following certain programming guidelines at service implementation time. Particularly, in an attempt to explain the root causes of most *understandability* and *discoverability* problems associated with services interfaces, in [6] a statistical correlation analysis between service implementation metrics and service interface bad practices occurrences has been reported. In this paper, we study the feasibility of obtaining more *maintainable* services by exploiting Object-Oriented metrics (OO) values from the source code implementing services. Similarly to [6], the approach in this work uses OO metrics as early indicators to guide software developers towards obtaining more maintainable services.

Interestingly, we have found that there is a statistically significant, high correlation between several traditional (source code-level) OO metrics and the catalog of (WSDL-level) service metrics described in [5]. This is the most comprehensive and rigorously evaluated catalog of metrics for measuring service maintainability from WSDL interfaces. A corollary of this finding is that software developers could consider applying simple early code refactorings to avoid obtaining non-maintainable services upon generating service descriptions. Although our findings do not depend on the programming language in which services are implemented, we focus on Java, which is widely used in back-end and hence service development. To evaluate our approach, we performed experiments with a data-set of 154 real services, and the most popular Java-to-WSDL mapping tool, i.e. Java2WSDL (`http://ws.apache.org/axis/java`).

The rest of the paper is organized as explained next. Section 2 provides the background necessary to understand the goals and results of our research. Section 3 surveys related works. Section 4 presents detailed analytical experiments that evidence the cor-

relation of OO metrics with the Web Service metrics proposed in [5]. Section 5 explains how this correlation can be exploted to predict and early improve service maintainability. Section 6 concludes the paper and describes future research opportunities.

2 Basic concepts

WSDL is a language that allows providers to describe two main aspects of a service, namely what it does (its functionality) and how to invoke it (its binding-related information). The former aspect reveals the functional service interface that is offered to potential consumers. The latter aspect specifies technological details, such as transport protocols and network addresses. Consumers use the former part to match third-party services against their needs, and the latter part to actually interact with the selected service. With WSDL, service functionality is described as a *port-type* $W = \{O_0(I_0, R_0), .., O_N(I_N, R_N)\}$, which lists one or more operations O_i that exchange input and return messages I_i and R_i, respectively. Port-types, operations and messages are labeled with unique names, and optionally they might contain some comments.

Messages consist of *parts* that transport data between providers and consumers of services, and vice-versa. Exchanged data is represented by using data-type definitions expressed in XML Schema Definition (XSD), a language to define the structure of an XML construct. XSD offers constructors for defining simple types (e.g. integer and string), restrictions, and both encapsulation and extension mechanisms to define complex constructs. XSD code might be included in a WSDL document using the *types* element, but alternatively it might be put into a separate file and imported from the WSDL document or external WSDL documents so as to achieve type reuse.

A requirement inherent to manually creating and manipulating WSDL and XSD definitions is that services are built in a *contract-first* manner, a methodology that encourages designers to first derive the WSDL interface of a service and then supply an implementation for it. However, the most used approach to build Web Services in the industry is *code-first*, which means that one first implements a service and then generates the corresponding service interface by automatically deriving the WSDL interface from the implemented code. This means that WSDL documents are not directly created by developers but are instead automatically derived via language-dependent tools. Such a tool performs a mapping T [6], formally $T : C \rightarrow W$.

T maps the main implementation class of a service ($C = \{M(I_0, R_0), .., M_N(I_N, R_N)\}$) to the WSDL document describing the service ($W = \{O_0(I_0, R_0), .., O_N(I_N, R_N)\}$). Then, T generates a WSDL document containing a port-type for the service implementation class, having as many operations O as public methods M the class defines. Moreover, each operation of W is associated with one input message I and another return message R, while each message comprises an XSD data-type representing the parameters of the corresponding class method. Tools like WSDL.exe, Java2WSDL, and gSOAP [7] rely on a mapping T for generating WSDLs from C#, Java and C++, respectively.

Fig. 1 shows the generation of a WSDL document using Java2WSDL. The mapping T in this case has associated each public method from the service code to an *operation* containing two *messages* in the WSDL document and these, in turn, are associated with an XSD data-type containing the parameters of that operation. Depending

on the tool used some minor differences between the generated WSDL documents may arise [6]. For instance, for the same service Java2WSDL generates only one port-type with all the operations of the Web Service, whereas WSDL.exe generates three port-types each bound to a different transport protocol.

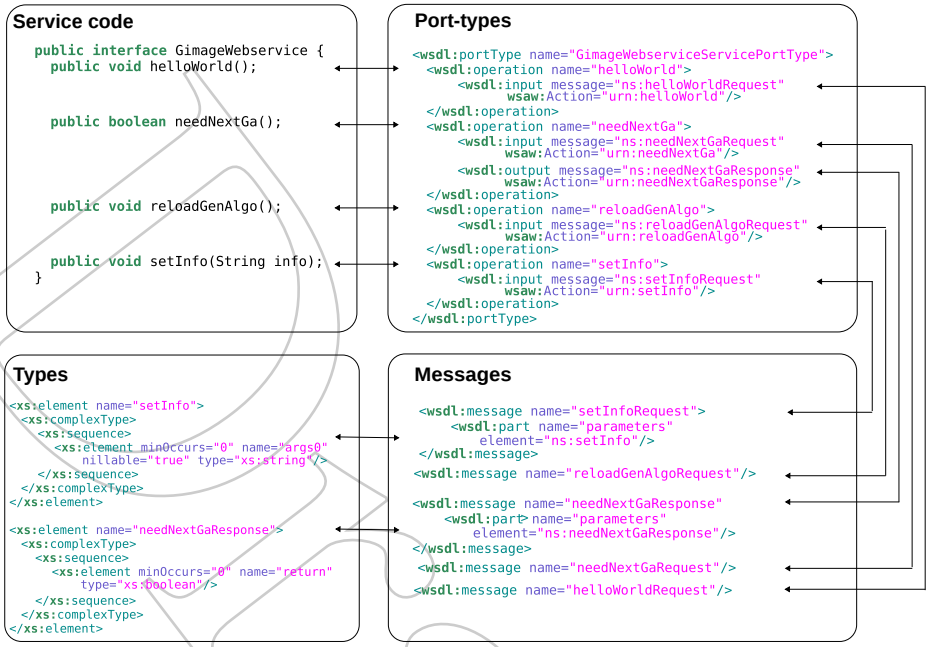

Fig. 1. WSDL generation in Java through the Java2WSDL tool

The fact supporting our hypothesis is precisely that WSDL metrics in the general sense are associated with API design attributes [6,4]. These latter have been throughly studied by the software engineering community and as a result suites of OO class-level metrics exist, such as the Chindamber and Kemerer's catalog [8]. Consequently, these metrics tell providers about how a service implementation conforms to specific design attributes. For example, the CBO (Coupling Between Objects) metric gives a hint on data model quality in terms of code development and *maintenance* facilities. Moreover, the LCOM (Lack of Cohesion Methods) metric measures how well the methods of a class are semantically related to each other, or the *cohesion* design attribute.

An interesting approach is then to assess whether a desired design attribute as measured by an appropriate OO metric is ensured after WSDL generation as measured by a WSDL-level metric suitable for the attribute (e.g. [5] when targeting maintainability or [3] when targeting complexity). As a corollary, by using well-known software metrics on a service code C, a service developer might have an estimation of how the resulting WSDL document W will be like in terms of maintainability and complexity since a known mapping T deterministically relates C with W. Then, based on these code

metric/WSDL metric relationships, it is possible to determine a wider range of metric values for C so that T generates W without undesirable WSDL-level metric values.

3 Related efforts

Although there has been a substantial amount of research to improve services interfaces quality [3,4,5], the approach to predict quality by basing on traditional OO metrics at development time remains rather unexplored. Indeed, this approach has been recently and exclusively explored in [6] to anticipate potential quality problems in services interfaces. Conceptually, the work presented in [6] studies the relationships between service implementation metrics and a key quality attribute of target services interfaces in WSDL, namely discoverability [4].

From services implementations the authors gathered 6 classic OO metrics, namely Chindamber and Kemerer's [8] CBO, LCOM, WMC (Weighted Methods Per Class), RFC (Response for Class), plus the CAM (Cohesion Among Methods of Class) metric from the work of Bansiya and Davis [9] and the well-known lines of code (LOC) metric. Additionally, they gathered 5 ad-hoc metrics, namely TPC (Total Parameter Count), APC (Average Parameter Count), ATC (Abstract Type Count), VTC (Void Type Count), and EPM (Empty Parameters Methods).

Regarding discoverability, the authors used a sub-set of the WSDL metrics listed in [4], which are focused on measuring aspects that affect discoverability, namely the legibility, conciseness, and understandability of WSDL descriptions [**?**,**?**,**?**]. Then, the authors collected a data-set of publicly available code-first Web Services projects, which by itself has been a valuable contribution to the field, and in turn analyzed the statistical relationship among metrics.

In the same direction that [6], this paper analyzes whether there are relations between service implementation metrics and the suite of metrics proposed by Baski and Misra [5], which comprises 4 novel metrics for measuring the complexity of the information exchanged by Web Services. As will be explained later, these metrics can be statically computed from a service interface in WSDL, since this metric suite is purely based on WSDL and XSD schema elements occurrences.

3.1 Data weight metric

Baski and Misra [5] defined the data complexity as "the complexity of data flowed to and from the interfaces of a Web service and can be characterized by an effort required to understand the structures of the messages that are responsible for exchanging and conveying the data". The definition of the Data Weight (DW) metric is based on the above, and computes the complexity of the data-types conveyed in services messages. To the sake of brevity, we will refer to the complexity of a message $C(m)$ as an indicator of the effort required to understand, extend, adapt, and test m, by basing on its structure. $C(m)$ counts how many elements, complex types, restrictions and simple types are exchanged by messages parts, as it is deeply explained in [5]. Formally:

$$DW(wsdl) = \sum_{i=1}^{n_m} C(m_i) \tag{1}$$

where n_m is the number of messages that the WSDL document exchanges. For the purposes of this paper, we have assumed n_m to consider only those messages that are linked to an offered operation of the WSDL document, thus it does not take into account dangling messages. The DM metric always returns a positive integer. The bigger the DM of a WSDL document, the more complex its operations messages are.

3.2 Distinct message ratio metric

The Distinct Message Ratio (DMR) metric complements DW by attenuating the impact of having similar-structured messages within a WSDL document. As the number of similar-structured messages increases the complexity of a WSDL document decreases, since it is easier to understand similar-structured messages than that of various-structured ones as a result of gained familiarity with repetitive messages [5]. Formally:

$$DMR(wsdl) = \frac{DMC(wsdl)}{n_m} \tag{2}$$

where the Distinct Message Count (DMC) metric can be defined as the number of distinct-structured messages represented by $[C(m), n_{args}]$ pair, i.e. the complexity value $C(m)$ and total number of arguments n_{args} that the message contains [5]. The DMR metric always returns a number in the range of [0,1], where 0 means that all defined messages are similar-structured, and 1 means that messages variety increases.

3.3 Message entropy metric

The Message Entropy (ME) metric exploits the probability of similar-structured messages to occur within a given WSDL document. Compared with the DMR metric, ME also bases on the fact that repetition of the same messages makes a developer more familiar with the WSDL document and results in ease of maintainability, but ME provides better differentiation among WSDL documents in terms of complexity. Formally:

$$ME(wsdl) = -\sum_{i=1}^{DMC(wsdl)} P(m_i) * log_2 P(m_i) \tag{3}$$

$$P(m_i) = \frac{nom_i}{n_m}$$

where nom_i is the number of occurrences of the i^{th} message, and in turn $P(m_i)$ represents the probability that such a message occurs within the given WSDL document. The ME metric outputs values greater or equal than zero. A low ME value shows that the messages are consistent in structure, which means that data complexity of a WSDL document is lower than that of the others having equal DMR values.

3.4 Message repetition scale metric

The Message Repetition Scale (MRS) metric analyses variety in structures of WSDL documents. By considering frequencies of $[C(m), n_{args}]$ pairs, MRS measures the consistency of messages as follows:

$$MRS(wsdl) = \sum_{i=1}^{DMC(wsdl)} \frac{nom_i^2}{n_m} \quad (4)$$

The possible values for MRS are in the range $1 \leq MRS \leq n_m$. When comparing two or more WSDL documents, a higher MRS and lower ME show that the developer makes less effort to understand the messages structures owing to the repetition of similar-structured messages.

4 Statistical correlation among services metrics

We used the Spearman's rank coefficient to analyze whether metrics taken on service implementations are correlated with metrics from their WSDL documents or not. Broadly, the experiment consisted on gathering OO metrics from real Web Services, calculating the service interface metrics of the previous section from WSDL documents, and analyzing the correlation among all pairs of metrics.

To perform the analysis, we employed a data-set of 154 WSDL documents from open source projects, which was the newest version available of the data-set described in [6] at the time of writing this article. All projects are written in Java, and were collected via the source code search engines Merobase, Exemplar and Google Code. Each project offers at least one Axis' Java2WSDL Web Service description. Each service within each project consists of the implementation code and dependency libraries needed for compiling and generating WSDL documents. All in all, the generated data-set provided the means to perform a significant evaluation in the sense that the different Web Service implementations came from real-life software developers.

Regarding the correlation model, the independent variables were the WMC, CBO, RFC, LCOM, CAM, TPC, APC, ATC, VTC, EPM, and LOC metrics. On the other hand, the dependent variables were the DW, DMC, DMR, ME and MRS metrics. Metrics recollection is an extremely sensitive task for this experiment, but also a task that would require a huge amount of time to be manually carried on. Therefore, all the employed metrics have been automatically gathered by using an extended version of the *ckjm* [10] tool and a software library to automate the recollection of the WSDL metrics (`http://code.google.com/p/wsdl-metrics-soc-course/`).

Table 1. Correlation between OO metrics and WSDL ones

Metric	WMC	CBO	RFC	LCOM	LOC	CAM	TPC	APC	GTC	VTC	EPM
DW	**0.47**	**0.80**	**0.47**	**0.47**	**0.47**	**-0.48**	**0.60**	**0.38**	**0.58**	0.05	-0.03
DMC	**0.82**	**0.53**	**0.82**	**0.82**	**0.82**	**-0.83**	**0.74**	0.14	**0.39**	**0.16**	**0.29**
DMR	**-0.71**	**0.22**	**-0.71**	**-0.71**	**-0.71**	**0.43**	**-0.37**	**0.34**	**0.25**	-0.10	**-0.36**
ME	**0.72**	**0.62**	**0.72**	**0.72**	**0.72**	**-0.79**	**0.68**	**0.18**	**0.45**	0.13	**0.18**
MRS	**0.80**	-0.13	**0.80**	**0.80**	**0.80**	**-0.54**	**0.49**	**-0.28**	**-0.18**	0.10	**0.37**

Table 1 and Fig. 2 show the existing relations between the OO metrics (independent variables) and the WSDL metrics (dependent variables). For denoting those coefficients that are statistically significant at the 5% level or p-value < 0.05, which is a common choice when performing statistical studies [11], we employed bold numbers in Table 1, and circles in Fig. 2. The sign of the correlation coefficients (+, -) are depicted using black and white circles, respectively. A positive (black) relation means that when the independent variable grows, the dependent variable grows too, and when the independent variable falls the dependent goes down as well. Instead, a negative (white) relation means that when independent variables grow, dependent ones fall, and vice versa. The absolute value, or correlation factor, indicates the intensiveness of the relation regardless of its sign. Graphically, the diameter of a circle represents a correlation factor, i.e. the bigger the correlation factor the bigger the diameter in Fig. 2.

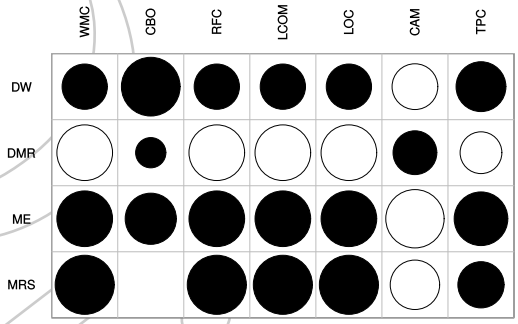

Fig. 2. Graphical representation of correlation analysis for rapidly identifying relations

5 A step towards early improving WSDL maintainability

By looking at Fig. 2 one could state that there is a high statistical correlation between the analyzed OO metrics and, at least, one metric for assessing the maintainability of services interfaces descriptions. Initially, this implies that every independent variable should be somehow "controlled" by software engineers attempting to obtain maintainable target WSDL documents. However, as determining the best set of *controllable* independent variables would deserve a deeper analysis within a longer paper, we will focus on determining a minimalist sub-set of OO metrics for this paper.

We employed two criteria for reducing the number of OO metrics. First, by basing on the study presented in [6] that shows the existence of groups of statistically dependent OO metrics, we select one representative metric for each group. In other words, if a group of variables in a data-set are strongly correlated, these variables are likely to measure the same underlying dimension (e.g. cohesion, complexity, or coupling). Table 2 shows the statistical correlation among OO metrics for the employed version of the data-set, which confirms that RFC, LCOM and LOC can be removed while keeping

WMC, since these are statistically correlated at the 5% level with correlation factor of 1.

Table 2. Correlation among OO metrics.

Metric	WMC	CBO	RFC	LCOM	LOC	CAM	TPC	APC	GTC	VTC	EPM
WMC	**1.00**	**0.20**	**1.00**	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	**0.28**	**0.41**
CBO	-	**1.00**	**0.20**	**0.20**	**0.20**	**-0.37**	**0.29**	**0.26**	**0.41**	-0.07	-0.15
RFC	-	-	**1.00**	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	**0.28**	**0.41**
LCOM	-	-	-	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	0.28	**0.41**
LOC	-	-	-	-	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	**0.28**	**0.41**
CAM	-	-	-	-	-	**1.00**	**-0.63**	0.08	**-0.24**	**-0.35**	**-0.36**
TPC	-	-	-	-	-	-	**1.00**	**0.55**	**0.33**	**0.28**	0.08
APC	-	-	-	-	-	-	-	**1.00**	**0.30**	0.04	**-0.33**
GTC	-	-	-	-	-	-	-	-	**1.00**	0.03	**-0.18**
VTC	-	-	-	-	-	-	-	-	-	**1.00**	**0.38**
EPM	-	-	-	-	-	-	-	-	-	-	**1.00**

Second, we removed the APC, GTC, VTC, and EPM metrics because they do not have at least one relationship with its correlation factor above |0.6| at the 5% level. The rationale of this criterion reduction was to keep only the highest correlated pairs of variables.

Table 3. Minimalist sub-set of correlated OO and WSDL metrics

Metric	WMC	CBO	RFC	LCOM	LOC	CAM	TPC
DW	-	**0.80**	-	-	-	-	**0.74**
DMC	**0.82**	-	**0.82**	**0.82**	**0.82**	**-0.83**	**0.68**
DMR	**-0.71**	-	**-0.71**	**-0.71**	**-0.71**	-	**0.60**
ME	**0.72**	**0.62**	**0.72**	**0.72**	**0.72**	**-0.79**	-
MRS	**0.80**	-	**0.80**	**0.80**	**0.80**	-	-

Table 3 represents a minimalist sub-set of correlated metrics, and shows that the DW metric depends on two OO metrics, i.e. CBO and TPC. Then, the DW of a service may be influenced by the number of classes coupled to its implementation, and by the number of parameters their operations exchange. The results also show that DW is not

highly influenced by cohesion related metrics, such as LCOM and CAM, neither by how many methods its implementation invokes (RFC) or methods complexity (WMC).

The DMC and ME metrics may be decreased by reducing the complexity of service implementation methods. This means that if a software developer modifies his/her service implementation and in turn this reduces the WMC, RFC and LOC metrics, such a fall will cause a decrement in DMC and ME. At the same time, DMC and ME may be reduced by improving the cohesion of services implementations. This is because when the cohesion of services implementations is improved, LCOM falls and CAM rises, which may produce a lower value for DMC and ME, by basing on the signs of their respective statistical relations. ME is influenced by CBO as well.

The DMR metric has negative correlations with WMC, RFC, and LOC. Surprisingly, this means that when the complexity of services implementations methods grows, the ratio of distinct messages falls. Also, DMR has a positive correlation with TPC, which means that the higher the number of operations parameters the higher the ratio of distinct messages. The MRS metric presents high correlations with 3 complexity and 1 cohesion service implementation metrics.

The presented evidence shows that pursuing an improvement in the DMR metric may conflict with other metric-driven goals. This means that if a software developer modifies his/her service implementation and in turn this produces an increment in the WMC, RFC, LOC or LCOM metrics, such an increment will cause that DMC, ME, and MRS alert about an increment in the WSDL document complexity, however the DMR will fall. Clearly, incrementing DMC, and ME would be undesirable, but this could be the side-effect of a gaining in the DMR metric. As in general software literature, this kind of situations could be treated as *trade-offs*, in which the software engineer should analyze and select among different metric-driven implementation alternatives.

6 Conclusions

We have empirically shown that when developing code-first Web Services, there is a high statistical correlation between several traditional code-level OO metrics and some WSDL-related metrics that measure maintainability at the WSDL level. This enforces the findings reported in [6], in which a correlation between some OO metrics and metrics that measure service discoverability was also found.

As discussed in Section 5, this would allow software engineers and developers to early adjust OO metrics, for example via M. Fowler's code refactorings, so that resulting WSDLs and hence services are more maintainable. Moreover, modern IDEs provide specific refactorings that can be employed to adjust the value of metrics such as WMC, CBO and RFC. For metrics which are not very popular among developers and therefore have not associated a refactoring (e.g. CAM), indirect refactorings may be performed. For example, CAM is negatively correlated to WMC (see Table 2), and hence refactoring for WMC means indirectly refactoring for CAM.

At present, we are generalizing our results by analyzing correlation for WSDL documents built via code-first tools other than Java2WSDL. Besides, we are studying the correlation of OO metrics and the WSDL metrics proposed by Harry Sneed [3], which is a comprehensive catalog of metrics for measuring Web Service complexity.

We are also planning to deeply study the aforementioned trade-offs so as to provide more decision support to engineers. Indeed, refactoring for a particular OO metric may yield good results as measured by some WSDL metrics but bad results with respect to other WSDL metrics in the same catalog. Preliminary experiments with the WSDL complexity metrics catalog presented in [3] also suggest that reducing the value of some OO metrics from Table 2 positively impacts on several complexity metrics, but at the same time negatively affects others. It is then necessary to determine to what extent these OO metrics are modified upon code refactoring so that a balance with respect to (ideally) all WSDL metrics in the catalog is achieved. This is however difficult specially when trying to balance the values of metrics of different catalogs, i.e. when attempting to build a service that is more maintainable according to Baski and Misra, but less complex according to Sneed.

Acknowledgments

We acknowledge the financial support provided by ANPCyT (PAE-PICT 2007-02311). We thank Martín Garriga for his predisposition and valuable help towards building the software tool that computes the studied Web Service maintainability metrics. We also thank Taiyun Wei, the author of the *R* script for drawing the correlation matrix of Fig. 2.

References

1. Juan Manuel Rodriguez, Marco Crasso, Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Bottom-up and top-down COBOL system migration to Web Services: An experience report. *IEEE Internet Computing*, 17(2):44–51, 2013.
2. Guadalupe Ortiz and Alfonso García De Prado. Improving device-aware Web Services and their mobile clients through an aspect-oriented, model-driven approach. *Information and Software Technology*, 52(10):1080–1093, 2010.
3. Harry M. Sneed. Measuring Web Service interfaces. In *12th IEEE International Symposium on Web Systems Evolution (WSE), 2010*, pages 111 –115, September 2010.
4. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.
5. D. Basci and S. Misra. Metrics suite for maintainability of extensible markup language Web Services. *IET Software*, 5(3):320–341, 2011.
6. Cristian Mateos, Marco Crasso, Alejandro Zunino, and José Luis Ordiales Coscia. Detecting WSDL bad practices in code-first Web Services. *International Journal of Web and Grid Services*, 7(4):357–387, 2011.
7. Robert A. Van Engelen and Kyle A. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 128–135. IEEE Computer Society, 2002.
8. S. Chidamber and C. Kemerer. A metrics suite for Object Oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
9. Jagdish Bansiya and Carl G. Davis. A hierarchical model for Object-Oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.
10. Diomidis Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22:9–11, 2005.
11. Stephen Stigler. Fisher and the 5% level. *Chance*, 21:12–12, 2008.