Estimating Web Service interface complexity and quality through conventional object-oriented metrics

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

[1] ISISTAN Research Institute.
`{mcrasso,cmateos,azunino}@conicet.gov.ar`
[2] UNICEN University.
`jlordiales@gmail.com`
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

Abstract. Historically, software engineers have conceived metric suites as valuable tools to estimate the quality of their software artifacts. Recently, a fresh computing paradigm called Service-Oriented Computing (SOC) has emerged at the crossing of massively distributed and heterogeneous software. This paper presents a statistical correlation analysis showing that classic software engineering metrics can be used to predict the complexity and quality of WSDL documents, the cornerstone software artifact when materializing this novel computing paradigm with Web-based technologies. For the experiments, 154 real world WSDL documents have been employed.

Keywords: SERVICE-ORIENTED COMPUTING; WEB SERVICES; CODE-FIRST; OBJECT-ORIENTED METRICS; EARLY DETECTION.

1 Introduction

The Service-Oriented Computing (SOC) paradigm has been steadily gaining territory in the software industry. With SOC, the composition of loosely coupled pieces of software, called *services*, drives software construction. A key to SOC is that services may be provided by third-parties who only expose services interfaces to the outer world.

This basic idea has been present in the software industry since a long time ago. However, the advances in distributed system technologies nowadays encourage engineers to implement the SOC paradigm in environments with higher levels of distribution and heterogeneity. For instance, broadband and ubiquitous connections enable to reach the Internet from everywhere and at every time, enabling a global scale marketplace of software services. In such a marketplace, providers may offer their services interfaces and consumers may invoke them regardless geographical aspects, using the current Web infrastructure as the communication channel. When services are implemented using standard Web languages and protocols –which is the most common scenario– they are called *Web Services*. Nowadays, Web Services are the leading character in diverse contexts. For example, they are the chosen technology usually employed when migrating legacy systems [13] or the technological protocol stack used when accessing remote information from smartphones [11].

Like programs or in general any other software artifact, service interface descriptions have a size, complexity and quality, all of which can be measured [16]. Previous research has emphasized on the importance of services interfaces and more specifically their non-functional concerns. Particularly, the work of [14] identifies common bad practices found in services interfaces, which impact on the understandability and discoverability of described services. In this context, understandability is the ability of a service interface description of being self-explanatory, i.e. a software engineer can effectively reason about a service purpose just by looking at its associated interface description. Discoverability, on the other hand, refers to the ability of a service of being easily retrieved, from a registry or repository, based on a partial description of its functionality, i.e. a query. At the same time, in [16] the author describes a suite of metrics to assess the complexity and quality of services interfaces.

In an attempt to approach the root causes of most understandability and discoverability problems associated with services interfaces, in [9] a statistical correlation analysis between service implementation metrics and service interface bad practices occurrences has been reported. The rationale behind the study is that in practice service interfaces are not build by hand, but instead they are automatically generated by mapping programming languages constructors onto service interface descriptions expressed in Web Service Definition Language (WSDL). WSDL is an XML-based format for describing a service as a set of operations, whose invocation is based on message exchange. For services implemented in Java, this mapping is usually achieved by tools such as Axis' Java2WSDL, Java2WS, EasyWSDL, and WSProvide.

As a complement, in this paper we study the feasibility of obtaining less *complex* and of the highest possible *quality* services by exploiting Object-Oriented metrics (OO) values from the code implementing services. Similar to [9], the approach behind this work relies on employing these metrics as early indicators to warn software engineers about the possibility of obtaining services with high complexity and low quality at development time. For the purposes of this paper, complexity analysis covers lingual, data, and structural aspects of WSDL documents, while quality analysis deals with modularity, adaptability, reusability, testability, portability and conformity quality attributes. Interestingly, we have found that there is a statistical significant, somewhat high correlation between several traditional (source code-level) OO metrics and the catalog of (WSDL-level) service metrics described in [16]. To date, this is the most comprehensive catalog of metrics for measuring service complexity and quality from only WSDL interfaces. All in all, we argue that based on these indicators, software engineers could consider applying simple early code refactorings to avoid obtaining services with the two mentioned problems upon generating service descriptions.

It is worth noting that although our approach to correlation does not depend on the programming language in which services are implemented, we limit the scope of our research to Java, which is very popular in back-end and hence service development. To evaluate our approach, we performed experiments by employing a data-set of real services, and one of the most popular Java-to-WSDL generation tool, namely Java2WSDL[4].

[4] `http://ws.apache.org/axis/java`

The rest of the paper is structured as follows. Section 2 gives some background necessary to understand the goals and results of the study presented in this paper. Section 3 surveys relevant related works. Section 4 presents detailed analytical experiments that evidence the correlation of OO metrics with the Web Service metrics proposed in [16]. Section 5 concludes the paper.

2 Background

WSDL allows providers to describe a service from two main angles, namely what it does (its functionality) and how to invoke it. Following the version 1.1 of the WSDL specification, the former part reveals the service interface that is offered to potential consumers. The latter part specifies technological aspects, such as transport protocols and network addresses. Consumers use the functional descriptions to match third-party services against their needs, and the technological details to invoke 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 arranges different operations O_i that exchange input and return messages I_i and R_i, respectively. Port-types, operations and messages must be labeled with unique names. Optionally, these WSDL elements might contain some documentation in the form of comments.

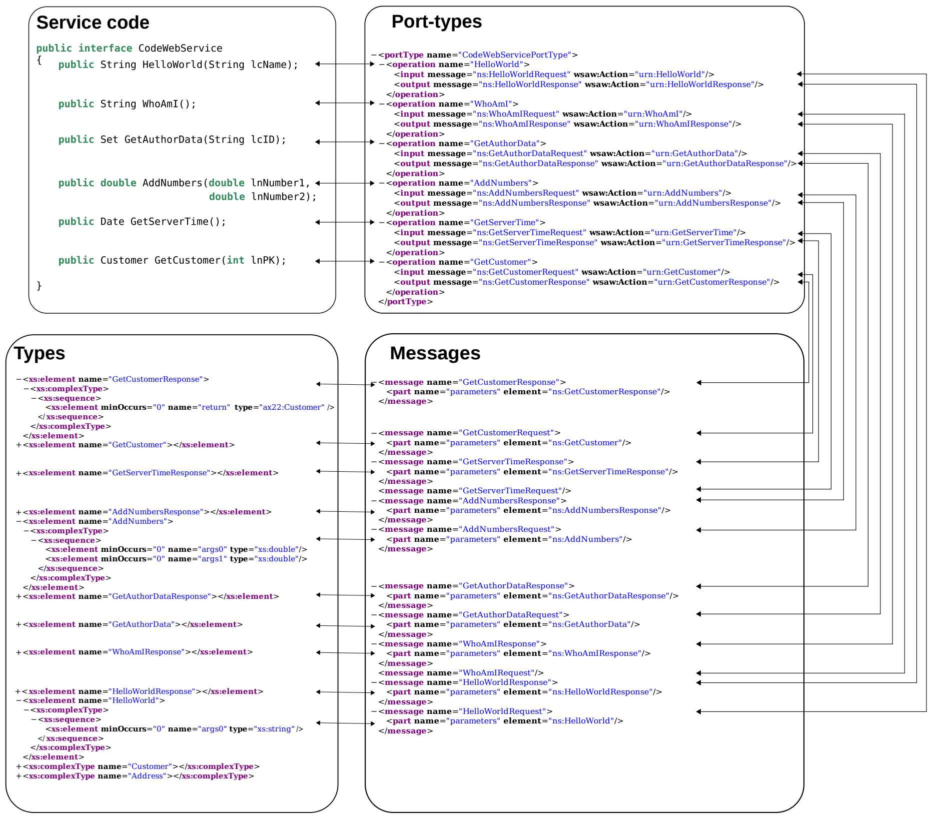
Messages consist of *parts* that transport data between providers and consumers of services, and vice-versa. Exchanged data is represented by using XML but according to specific data-type definitions in XML Schema Definition (XSD), a language to define the structure of an XML construct. XSD offers constructors 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 in order 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 by 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 humans but are instead automatically derived via language-dependent tools. Such a tool performs a mapping T [9], formally:

$$T : C \rightarrow W, \tag{1}$$

Mapping T from $C = \{M(I_0, R_0), .., M_N(I_N, R_N)\}$ or the front-end class implementing a service to $W = \{O_0(I_0, R_0), .., O_N(I_N, R_N)\}$ or the WSDL document describing the service, precisely generates a WSDL document containing a port-type for the service implementation class, having as many operations O as public methods M are defined in the class. Moreover, each operation of W is associated with one input message I and another return message R, while each message conveys an XSD type that stands for the parameters of the corresponding class method. Tools like WSDL.exe, Java2WSDL, and gSOAP [21] are based on a mapping T for generating WSDL documents from C#, Java and C++, respectively, though each tool implements T in a particular manner mostly because of the different characteristics of the involved programming languages.

Figure 1 shows the generation of a WSDL document using Java2WSDL. It can be noted that the mapping T maps 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 type containing the parameters of that operation. As shown in [9], depending on the tool used, however, some minor differences between the generated WSDL documents may arise. 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 (one for each transport protocol). As we mentioned before, these differences are a result of the implementation each tool uses when applying the mapping to the service code.



```
public interface CodeWebService
{   public String HelloWorld(String lcName);

    public String WhoAmI();

    public Set GetAuthorData(String lcID);

    public double AddNumbers(double lnNumber1,
                             double lnNumber2);

    public Date GetServerTime();

    public Customer GetCustomer(int lnPK);

}
```

```
-<portType name="CodeWebServicePortType">
 -<operation name="HelloWorld">
    <input message="ns:HelloWorldRequest" wsaw:Action="urn:HelloWorld"/>
    <output message="ns:HelloWorldResponse" wsaw:Action="urn:HelloWorldResponse"/>
  </operation>
 -<operation name="WhoAmI">
    <input message="ns:WhoAmIRequest" wsaw:Action="urn:WhoAmI"/>
    <output message="ns:WhoAmIResponse" wsaw:Action="urn:WhoAmIResponse"/>
  </operation>
 -<operation name="GetAuthorData">
    <input message="ns:GetAuthorDataRequest" wsaw:Action="urn:GetAuthorData"/>
    <output message="ns:GetAuthorDataResponse" wsaw:Action="urn:GetAuthorDataResponse"/>
  </operation>
 -<operation name="AddNumbers">
    <input message="ns:AddNumbersRequest" wsaw:Action="urn:AddNumbers"/>
    <output message="ns:AddNumbersResponse" wsaw:Action="urn:AddNumbersResponse"/>
  </operation>
 -<operation name="GetServerTime">
    <input message="ns:GetServerTimeRequest" wsaw:Action="urn:GetServerTime"/>
    <output message="ns:GetServerTimeResponse" wsaw:Action="urn:GetServerTimeResponse"/>
  </operation>
 -<operation name="GetCustomer">
    <input message="ns:GetCustomerRequest" wsaw:Action="urn:GetCustomer"/>
    <output message="ns:GetCustomerResponse" wsaw:Action="urn:GetCustomerResponse"/>
  </operation>
 </portType>
```

```
-<xs:element name="GetCustomerResponse">
 -<xs:complexType>
  -<xs:sequence>
     <xs:element minOccurs="0" name="return" type="ax22:Customer" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>
+<xs:element name="GetCustomer"></xs:element>

+<xs:element name="GetServerTimeResponse"></xs:element>

+<xs:element name="AddNumbersResponse"></xs:element>
-<xs:element name="AddNumbers">
 -<xs:complexType>
  -<xs:sequence>
     <xs:element minOccurs="0" name="args0" type="xs:double"/>
     <xs:element minOccurs="0" name="args1" type="xs:double"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
+<xs:element name="GetAuthorDataResponse"></xs:element>

+<xs:element name="GetAuthorData"></xs:element>

+<xs:element name="WhoAmIResponse"></xs:element>

+<xs:element name="HelloWorldResponse"></xs:element>
-<xs:element name="HelloWorld">
 -<xs:complexType>
  -<xs:sequence>
     <xs:element minOccurs="0" name="args0" type="xs:string" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>
+<xs:complexType name="Customer"></xs:complexType>
+<xs:complexType name="Address"></xs:complexType>
```

```
-<message name="GetCustomerResponse">
   <part name="parameters" element="ns:GetCustomerResponse"/>
 </message>

-<message name="GetCustomerRequest">
   <part name="parameters" element="ns:GetCustomer"/>
 </message>
-<message name="GetServerTimeResponse">
   <part name="parameters" element="ns:GetServerTimeResponse"/>
 </message>
 <message name="GetServerTimeRequest"/>
-<message name="AddNumbersResponse">
   <part name="parameters" element="ns:AddNumbersResponse"/>
 </message>
-<message name="AddNumbersRequest">
   <part name="parameters" element="ns:AddNumbers"/>
 </message>

-<message name="GetAuthorDataResponse">
   <part name="parameters" element="ns:GetAuthorDataResponse"/>
 </message>
-<message name="GetAuthorDataRequest">
   <part name="parameters" element="ns:GetAuthorData"/>
 </message>
-<message name="WhoAmIResponse">
   <part name="parameters" element="ns:WhoAmIResponse"/>
 </message>
 <message name="WhoAmIRequest"/>
-<message name="HelloWorldResponse">
   <part name="parameters" element="ns:HelloWorldResponse"/>
 </message>
-<message name="HelloWorldRequest">
   <part name="parameters" element="ns:HelloWorld"/>
 </message>
```

Fig. 1. WSDL generation in Java

Precisely, the fact underpinning our study is that WSDL metrics in the general sense are strongly associated with API design attributes [9,14]. 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 metric catalog [5]. Consequently, these metrics tell providers about how a service implementation conforms to specific design attributes. For example, the LCOM (Lack of Cohesion Methods) metric provides a mean to measure 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 maintained after WSDL generation as measured by a WSDL-level metric suitable for the attribute (e.g. [16] when targeting complexity or quality). As a corollary, by employing well-known software engineering metrics on a service code C, a provider might have an estimation of how the resulting WSDL document W will be like in terms of understandability, discoverability, complexity and quality, since a known mapping T relates C with W. If indeed such code metric/anti-pattern relationships exist, then it would be possible to determine a wider range of metric values for C so that T generates W without undesirable WSDL-level metric values in the best case.

3 Related work

It can be found in the literature that researchers have been using traditional OO metrics at development time of conventional software to predict the number of bugs or other quality attributes [19,7,10]. With respect to Web Services, though there has been a substantial amount of research concerning improving services interfaces quality [16,14,3], the approach to associate implementation metrics with services interfaces has been recently and exclusively explored in [9].

In [9] the authors collect a publicly available data-set of Web Services projects, which by itself has been a valuable contribution to the field, and analyze the statistical relationships between metrics that were taken from services implementations and metrics taken from corresponding WSDL documents. In particular, the authors employed a sub-set of the WSDL metrics presented in [14], as it subsumes those research works focused on obtaining more legible, clear and discoverable service WSDL descriptions [6,4,12]. Therefore, [9] studies the relationships between service implementation metrics and the discoverability of target services interfaces in WSDL.

As will be explained in the next section this paper analyzes whether there are relations between service implementation metrics and the suite of metrics proposed by H. Sneed [16], which broadly comprises metrics to assess the complexity of services interfaces and to determine their size for estimating testing effort. The suite consists of different kinds of metrics, which range from common size measurements like lines of code and number of statements, to metrics for measuring the complexity and quality of Web Services. All the involved metrics can be statically computed from a service interface in WSDL, since the metric suite is purely based on WSDL schema elements occurrences. For the purposes of this work, we focus on two groups of metrics, namely the 6 complexity metrics: Interface Data Complexity, Interface Relation Complexity, Interface Format Complexity, Interface Structure Complexity, Data Flow Complexity (Elshof's Metric), Language Complexity (Halstead's Metric), and the 6 quality metrics: Modularity, Adaptability, Reusability, Testability, Portability, and Conformity. It is worth noting that the terms used to denote quality metrics are known in the literature as properties, factors or characteristics [8]. However, we are here strictly following the terminology of Harry S. Sneed, which proposed these quality metrics.

Additionally, we employed two more metrics that represent the average of either previous groups, namely Average Interface Complexity and Average Interface Quality. Metrics results are expressed as a value on a scale of 0 to 1. For complexity metrics, 0 to 0.4 indicates low complexity, 0.4 to 0.6 indicates average complexity, 0.6 to 0.8 indicates high complexity and over 0.8 indicates that the code is not well designed. Instead, for quality metrics 0 to 0.2 indicates no quality, 0.2 to 0.4 indicates low quality, 0.4 to 0.6 indicates median quality, 0.6 to 0.8 indicates high quality, and 0.8 to 1.0 indicates very high quality [15].

The set of rules employed for the Conformity metric has been extrapolated from the work of [20], and consists of 12 rules, namely Adherence to the current standard, Use of prescribed name spaces, Use of in source documentation, Avoidance of any data type, Adherence to the naming convention, Hiding of elementary data types, Restricting the size of data groups, Limiting interface width, Enforcing the first normal form, Minimizing the number of requests, Not exceeding a give size limit, and Limiting the use of links.

The next section presents the performed statistical analysis.

4 Statistical analysis and experiments

We used regression and correlation methods to analyze whether OO metrics taken on service implementations are correlated with metrics from the associated WSDL documents or not. Broadly, the experiment consisted on gathering OO metrics from open source Web Services, calculating Sneed's service interface complexity and quality metrics from WSDL documents, and analyzing the correlation among all pairs of metrics. To perform the analysis, we employed the newest version available of the data-set described in [9] at the time of writing this article. This data-set consists of 154 WSDL documents from open source projects, which were collected via the Merobase component filter, the Exemplar engine and the Google Code Web site. All projects are written in Java, and each project offers at least one Axis' Java2WSDL Web Service description. The data-set is self-contained in the sense that for each service, its implementation code and dependency libraries needed for compiling and generating WSDL documents are in the data-set. 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 engineers. For space reasons, most of the rest of the core details of the statistical experiment shown in this section can be found in [9] as well.

We used 11 metrics for measuring services implementations, which played the role of independent variables. WMC, CBO, RFC, and LCOM have been selected from the work of Chindamber and Kemerer [5]. The WMC (Weighted Methods Per Class) metric counts the methods of a class. CBO (Coupling Between Objects) counts how many methods or instance variables defined by other classes are accessed by a given class. RFC (Response for Class) counts the methods that can potentially be executed in response to a message received by an object of a given class. LCOM (Lack of Cohesion Methods) provides a mean to measure how well the methods of a class are related to each other. From the work of Bansiya and Davis [2] we picked CAM (Cohesion Among Methods of Class) metric. CAM computes the relatedness among methods based upon the parameter list of these methods. Additionally, we used a number of ad-hoc measures we thought could be related to the WSDL metrics, namely TPC (Total Parameter Count), APC (Average Parameter Count), ATC (Abstract Type Count), VTC (Void Type Count), and EPM (Empty Parameters Methods). The last employed metric was the well-known lines of code (LOC) metric.

On the other hand, the dependent variables were the metrics at the WSDL-level, i.e. those proposed in [16]. Table 1 statistically describes both independent and dependent variables. As shown in Std. Dev. column all dependent variables values were concentrated around the central tendency (i.e. Mean column), while this phenomenon persisted for independent variables values with some exceptions namely LCOM, WMC, and TPC. The Skewness column characterizes the location for metrics values distribution, in particular these results showed that most distributions are skewed to the right since their means were higher than their median. Furthermore, the skewness of three metrics was very near to 0, which is the skewness of the Normal distribution. The Kurtosis column shows that most metrics values did not tend to have a flat top near the mean rather than a distinct peak near the mean, decline rather rapidly, and have heavy tails. These results and the correlation analysis were obtained by using Apache's Commons Math library[5].

It is worth noting that all the employed OO metrics have been automatically gathered by using an extended version of the *ckjm* [17] tool. At the same time, we employed Sneed's SoftAUDIT tool [16] to automate the recollection of the WSDL metrics. SoftAUDIT receives a given WSDL document as input, and returns a metric report in a structured text file. 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. As a proof of this, we randomly selected a hand-full of Web Services projects from the data-set, and measured the averaged time needed for manually analyzing them, which was 2 days/developer. Besides being a time consuming task, it was an error prone task. Finally, it is worth noting that the data-set used in the experiments is available upon request. The next section presents the correlation analysis results.

4.1 Correlation analysis between OO and WSDL metrics

We used the Spearman's rank correlation coefficient in order to establish the existing relations between the two kind of variables of our model, i.e. the OO metrics (independent variables) and the WSDL metrics (dependent variables). Table 2 shows the correlation between the employed OO and WSDL metrics.

The cell values in bold are those coefficients which are statistically significant at the 5% level, i.e. p-value < 0.05, which is a common choice when performing statistical studies [18]. The sign of the correlation coefficients defines the direction of the relationship, i.e. positive or negative. A positive relation means that when the independent

[5] Apache's Commons Math library, `http://commons.apache.org/math`

Table 1. Descriptive statistics

Metric	**Min**	**Max**	**Mean**	**Std. Dev.**	**Kurtosis**	**Skewness**
Interface Data Complexity	0.10	0.90	0.82	0.09	27.91	-4.53
Interface Relation Complexity	0.10	0.90	0.88	0.13	34.52	-6.00
Interface Format Complexity	0.10	0.90	0.23	0.14	14.18	3.34
Interface Structure Complexity	0.10	0.75	0.13	0.08	28.21	5.10
Data Flow Complexity (Elshof's Metric)	0.10	0.90	0.23	0.07	51.04	6.18
Language Complexity (Halstead's Metric)	0.16	0.87	0.19	0.06	81.00	8.02
Average Interface Complexity	0.38	0.61	0.41	0.02	40.55	5.26
Interface Modularity	0.10	0.82	0.14	0.12	27.72	5.26
Interface Adaptability	0.10	0.90	0.60	0.30	-1.23	-0.49
Interface Reusability	0.10	0.90	0.31	0.13	9.97	1.93
Interface Testability	0.10	0.90	0.12	0.13	33.94	5.93
Interface Portability	0.33	0.50	0.49	0.03	33.70	-5.90
Interface Conformity	0.79	1.00	0.92	0.03	5.37	-0.41
Average Interface Quality	0.33	0.68	0.43	0.06	4.61	1.34
WMC	1.00	97.00	5.73	11.14	35.87	5.47
CBO	0.00	27.00	2.03	2.92	37.60	5.05
RFC	1.00	97.00	5.73	11.14	35.87	5.47
LCOM	0.00	4656.00	75.21	427.43	90.28	9.00
LOC	1.00	97.00	5.73	11.14	35.87	5.47
CAM	0.13	1.00	0.78	0.23	-0.86	-0.61
TPC	0.00	228.00	10.92	24.23	46.77	6.16
APC	0.00	17.00	2.04	1.83	30.61	4.45
GTC	0.00	20.00	1.09	2.26	34.15	5.01
VTC	0.00	25.00	1.05	3.54	26.65	4.96
EPM	0.00	11.00	0.57	1.64	16.52	3.85

Table 2. Correlation between OO metrics and WSDL ones

Metric	**WMC**	**CBO**	**RFC**	**LCOM**	**LOC**	**CAM**	**TPC**	**APC**	**GTC**	**VTC**	**EPM**
Interface Data Complexity	**-0.83**	**-0.23**	**-0.83**	**-0.83**	**-0.83**	**0.64**	**-0.61**	0.12	-0.12	-0.01	**-0.22**
Interface Relation Complexity	**-0.53**	**0.28**	**-0.53**	**-0.53**	**-0.53**	**0.21**	**-0.44**	0.01	0.12	-0.12	**-0.22**
Interface Format Complexity	0.07	**-0.40**	0.07	0.07	0.07	-0.00	-0.09	**-0.27**	**-0.38**	**0.44**	**0.27**
Interface Structure Complexity	**0.83**	0.01	**0.83**	**0.83**	**0.83**	**-0.62**	**0.57**	**-0.17**	-0.04	0.15	**0.37**
Data Flow Complexity (Elshof's metric)	**0.75**	0.08	**0.75**	**0.75**	**0.75**	**-0.54**	**0.63**	0.04	**0.19**	-0.01	0.15
Language Complexity (Halstead's metric)	0.09	**0.70**	0.09	0.09	0.09	-0.12	**0.29**	**0.40**	**0.50**	**-0.29**	**-0.25**
Average Interface Complexity	-0.03	**-0.28**	-0.03	-0.03	-0.03	0.06	-0.07	-0.07	-0.15	**0.32**	**0.16**
Interface Modularity	**0.38**	**-0.57**	**0.38**	**0.38**	**0.38**	**-0.17**	0.03	**-0.43**	**-0.45**	**0.22**	**0.56**
Interface Adaptability	**0.61**	**-0.45**	**0.61**	**0.61**	**0.61**	**-0.41**	**0.27**	**-0.37**	**-0.17**	0.15	**0.33**
Interface Reusability	**-0.60**	0.12	**-0.60**	**-0.60**	**-0.60**	**0.55**	**-0.48**	0.04	-0.16	**-0.50**	**-0.24**
Interface Testability	**0.23**	**0.20**	**0.23**	**0.23**	**0.23**	**-0.25**	**0.23**	0.09	-0.09	**0.28**	**0.27**
Interface Portability	**0.57**	**0.38**	**0.57**	**0.57**	**0.57**	**-0.49**	**0.56**	**0.17**	**0.37**	-0.13	0.05
Interface Conformity	-0.05	**0.62**	-0.05	-0.05	-0.05	0.05	**0.22**	**0.46**	**0.50**	-0.12	**-0.30**
Average Interface Quality	**0.69**	**-0.33**	**0.69**	**0.69**	**0.69**	**-0.45**	**0.36**	**-0.33**	**-0.22**	0.07	**0.41**

variable grows, the dependent variable grows too, and when the independent variable falls the dependent goes down as well. Instead, a negative relation means that when independent variables grow, the dependent metrics fall, and vice versa. The absolute value, or correlation factor, indicates the intensiveness of the relation regardless of its sign.

Additionally, for the sake of readability, we have employed a different approach to depict the correlation matrix, which is shown in Fig. 2. In the Figure, blank cells stand for not statistically significant correlations, whereas cells with circles represent correlation factors at the 5% level. The diameter of a circle represents a correlation factor, i.e. the bigger the correlation factor the bigger the diameter. The color of a circle stands for the correlation sign, being black used for positive correlations and white for negative ones.

4.2 Exploiting correlation results for improving WSDL documents

From both Table 2 and Fig. 2, it can be observed that there is a somewhat high statistical correlation between a sub-set of the analyzed OO metrics and some of the Sneed's WSDL metrics. This fact in principle allows us to determine which OO metrics could be somehow "controlled" by a software engineer attempting to assure the complexity and quality of his/her target WSDL documents. However, for the scope of this paper we will focus on determining a minimalist sub-set of OO metrics, because determining the best set of metrics would deserve a deeper analysis within a longer paper.

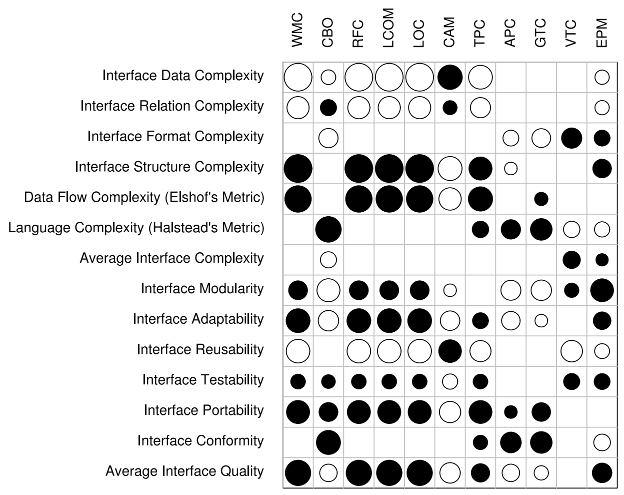


Fig. 2. Graphical representation of the correlation between OO metrics and WSDL ones

Since all OO metrics correlate to at least one WSDL metric, one could argue that every independent variable should be *controllable*. However, the study presented in [9] shows that some OO metrics are not statistically independent among them and, therefore, capture redundant information. 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 (i.e. cohesion, complexity, coupling, etc.). Because of this, we also calculated the statistical correlation among OO metrics for the employed version of the data-set. Table 3 shows the results, which confirm that OO metrics are not independent between each other. Then, RFC, LCOM and LOC can be removed while keeping WMC, since these are statistically correlated at the 5% level with the maximum correlation factor. To understand why this happens it is worth considering that, in most cases, the WSDL generation tool input is a code facade containing only the signatures of the services' operations. Therefore, each new method added to the facade contributes with exactly one new line of code, which explains the perfect correlation between WMC and LOC. Similarly, since LCOM measures the cohesion of a class based on its instance variables but the services' facades contain no such variables, the value of these metrics increases in the same proportion as WMC does. Finally, RFC is defined as the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class. This set includes the methods in the class and methods that can be invoked on other objects. Since methods signatures contain no logic, no invocations are made from the class methods. Therefore, for these facades, the value of RFC is exactly the same of WMC, thus resulting in a maximum correlation factor between the two.

Returning to the correlation between OO and WSDL metrics, two WSDL metrics base on other WSDL metrics. Average Interface Complexity is the average value of complexity metrics, and Average Interface Quality is the average value of quality metrics. Then, these two metrics may be removed by basing on the fact that their statistical relationships are represented by base metrics.

Table 3. 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	**0.20**	**1.00**	**0.20**	**0.20**	**0.20**	**-0.37**	**0.29**	**0.26**	**0.41**	-0.07	-0.15
RFC	**1.00**	**0.20**	**1.00**	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	**0.28**	**0.41**
LCOM	**1.00**	**0.20**	**1.00**	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	0.28	**0.41**
LOC	**1.00**	**0.20**	**1.00**	**1.00**	**1.00**	**-0.84**	**0.76**	-0.07	**0.17**	**0.28**	**0.41**
CAM	**-0.84**	**-0.37**	**-0.84**	**-0.84**	**-0.84**	**1.00**	**-0.63**	0.08	**-0.24**	**-0.35**	**-0.36**
TPC	**0.76**	**0.29**	**0.76**	**0.76**	**0.76**	**-0.63**	**1.00**	**0.55**	**0.33**	**0.28**	0.08
APC	-0.07	**0.26**	-0.07	-0.07	-0.07	0.08	**0.55**	**1.00**	**0.30**	0.04	**-0.33**
GTC	**0.17**	**0.41**	**0.17**	**0.17**	**0.17**	**-0.24**	**0.33**	**0.30**	**1.00**	0.03	**-0.18**
VTC	**0.28**	-0.07	**0.28**	**0.28**	**0.28**	**-0.35**	**0.28**	0.04	0.03	**1.00**	**0.38**
EPM	**0.41**	-0.15	**0.41**	**0.41**	**0.41**	**-0.36**	0.08	**-0.33**	**-0.18**	**0.38**	**1.00**

Furthermore, if we want to keep only highest correlated pairs of variables, the correlation factors below |0.6| at the 5% level can be removed. Then, the OO metrics APC, GTC, and VTC can be removed. Therefore, the 11x14 Table 2 may be reduced into a new 4x7 table (see Table 4), which represents a minimalist sub-set of correlated metrics.

Table 4. Minimalist sub-set of correlated OO and WSDL metrics

Metric	WMC	CBO	CAM	TPC
Interface Conformity	-	**0.62**	-	-
Interface Data Complexity	**-0.83**	-	**0.64**	**-0.61**
Interface Structure Complexity	**0.83**	-	**-0.62**	-
Data Flow Complexity (Elshof's metric)	**0.75**	-	-	**0.63**
Language Complexity (Halstead's metric)	-	**0.70**	-	-
Interface Adaptability	**0.61**	-	-	-
Interface Reusability	**-0.60**	-	-	-

In order to analyze the minimalist sub-set shown in Table 4, it is worth remembering the importance of correlation factor signs, since the higher the value of a WSDL metric the higher its complexity or the best its quality is. To clarify this, let us take for example the case of the TPC metric, which is positively correlated to Data Flow Complexity (Elshof's metric), but negatively correlated to Interface Data Complexity. This means that if a software engineer modifies his/her service implementation and in turn this produces an increment in the TPC metric, such an increment will cause an increment in Data Flow Complexity (Elshof's metric), but a fall in Interface Data Complexity. Clearly, incrementing Data Flow Complexity (Elshof's metric) would be undesirable. However the presented evidence shows that this could be the side-effect of pursuing an improvement in the Interface Data Complexity metric. From now on, we will refer to this kind of situations as *trade-offs*. As in software literature in general, here a trade-off represents a situation in which the software engineer should analyze and select among different metric-driven implementation alternatives.

As shown in Table 4, the resulting OO metrics are correlated with at least two WSDL metrics, and the signs of the correlations within one metric are always opposite, with the exception of the CBO metric. Therefore, some metrics represent trade-off opportunities. For example, the CAM metric is negatively related to Interface Structure Complexity, but positively related to Interface Data Complexity. Then, by incrementing the CAM metric, resulting WSDL documents will present a better value for Interface Structure Complexity metric than the original WSDL document. However, this will cause a deterioration of Interface Data Complexity metric, since this latter and CAM statistically grow together.

Controlling the CBO metric is safe, in the sense that it does not present trade-off situations and by modifying CBO no undesired collateral effects will be generated. This metric is positively correlated to Interface Conformity and Interface Data Complexity. Therefore, a WSDL document may be improved in terms of conformity and data complexity alike by decrementing the CBO metric from its associated service implementation.

Finally, the metric WMC is correlated to 5 of the 7 WSDL metrics in Table 4. The correlation factor signs indicate that the metric is positively related to Interface Adaptability, Interface Structure Complexity, and Data Flow Complexity. This means that by decrementing WMC on a service implementation, the corresponding WSDL document may present improvements with regard to Interface Adaptability, Interface Structure Complexity, and Data Flow Complexity metrics. However, there are two negative correlations, i.e. with Interface Reusability and Interface Data Complexity metrics, respectively. Therefore, the WMC present different 3x2 trade-offs.

5 Conclusions

In this article, we have empirically shown that when developing code-first Web Services, there is a significant statistical correlation between several traditional OO metrics and some WSDL-related metrics that measure complexity and quality at the service interface (i.e. WSDL) level. It is worth noting that even when a correlation between two variables does not imply causation, it is a necessary condition for it [1]. In other words, we are not stating that certain levels of WSDL quality are strictly caused by OO metrics values in their associated implementations, but there is indeed a significant correlation between source code-level and WSDL-level metrics. This enforces the findings reported in [9], in which a correlation between some OO metrics and metrics that measure service discoverability was also found. Interestingly, these facts allow software engineers to early adjust OO metrics, for example via M. Fowler's code refactorings, so that resulting WSDL and hence services are less complex, and of better quality.

However, these findings open the door to extra research questions related to how to increase/reduce a single OO metric, and how to deal with the various trade-offs that arise. With respect to the former issue, 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 that popular among developers/IDEs and therefore have not associated a refactoring (e.g. CAM), indirect refactorings may be performed. For example, from Table 3, CAM is negatively correlated to WMC, so refactoring for WMC means indirect refactoring for CAM. In principle, a complete catalog of refactoring actions for each metric regardless popularity levels thus could be built.

The trade-off problem is on the other hand largely more challenging as we have found that 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 target catalog. For example, for the metrics catalog of [16], it can be seen from Table 4 that reducing the WMC metric positively impacts on Interface Reusability, but negatively affects Interface Structure Complexity. However, the absolute correlation between WMC and Interface Reusability is weaker than the correlation between WMC and Interface Structure Complexity (i.e. 0.60 against 0.83). It is then necessary to determine to what extent WMC is modified upon code refactoring so that a balance with respect to (ideally) all WSDL metrics in a 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 of better quality, and lower complexity, but also discoverable. Therefore, addressing this issue is subject of further research.

In a follow-up study, we are also planning to generalize and extend our results by analyzing correlation when WSDL documents are generated via code-first tools other than Java2WSDL, particularly Java2WS, EasyWSDL, and WSProvide. This will allow us to assess the effect of the associated code-WSDL mappings in the correlation tables reported so far. Lastly, we will study the correlation of OO metrics and fresh WSDL metrics, particularly the work by Baski & Misra [3], which proposes a comprehensive catalog of metrics for measuring Web Service maintainability.

Acknowledgments

We acknowledge the financial support provided by ANPCyT (PAE-PICT 2007-02311). We deeply thank Harry M. Sneed for his predisposition and valuable help towards building this article. He provided us with several relevant documents and articles and the SoftAUDIT tool used in our experiments. We also thank Taiyun Wei, the author of the *R* script for drawing the correlation matrix of Fig. 2.

References

1. Aldrich, J.: Correlations genuine and spurious in pearson and yule. Statistical Science (10), 364–376 (1995)

2. Bansiya, J., Davis, C.G.: A hierarchical model for Object-Oriented design quality assessment. IEEE Transactions on Software Engineering 28, 4–17 (January 2002)
3. Baski, D., Misra, S.: Metrics suite for maintainability of extensible markup language Web Services. IET Software 5(3), 320–341 (2011)
4. Blake, M.B., Nowlan, M.F.: Taming Web Services from the wild. IEEE Internet Computing 12, 62–69 (September 2008)
5. Chidamber, S., Kemerer, C.: A metrics suite for Object Oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
6. Fan, J., Kambhampati, S.: A snapshot of public Web Services. SIGMOD Record 34(1), 24–32 (2005)
7. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of Object-Oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering 31(10), 897–910 (2005)
8. IEEE Computer Society: 610.12-1990 - IEEE standard glossary of software engineering terminology. `http://standards.ieee.org/findstds/standard/610.12-1990.html` (1990)
9. Mateos, C., Crasso, M., Zunino, A., Coscia, J.L.O.: Detecting WSDL bad practices in code-first Web Services. International Journal of Web and Grid Services 7(4) (2011), to appear
10. Meirelles, P., Jr., C.S., Miranda, J., Kon, F., Terceiro, A., Chavez, C.: A study of the relationships between source code metrics and attractiveness in free software projects. In: Brazilian Symposium on Software Engineering (SBES '10). vol. 0, pp. 11–20. IEEE Computer Society, Los Alamitos, CA, USA (2010)
11. Ortiz, G., Prado, A.G.D.: 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)
12. Pasley, J.: Avoid XML schema wildcards for Web Service interfaces. IEEE Internet Computing 10, 72–79 (2006)
13. Rodriguez, J.M., Crasso, M., Mateos, C., Zunino, A., Campo, M.: Bottom-up and top-down COBOL system migration to Web Services: An experience report. IEEE Internet Computing (2011), to appear
14. Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Improving Web Service descriptions for effective service discovery. Science of Computer Programming 75(11), 1001–1021 (2010)
15. Sneed, H.M.: Applying size complexity and quality metrics to an Object-Oriented application. In: Proceedings of the European Software Control and Metrics Conference (1999)
16. Sneed, H.M.: Measuring Web Service interfaces. In: 12th IEEE International Symposium on Web Systems Evolution (WSE), 2010. pp. 111 –115 (Sep 2010)
17. Spinellis, D.: Tool writing: A forgotten art? IEEE Software 22, 9–11 (2005)
18. Stigler, S.: Fisher and the 5% level. Chance 21, 12–12 (2008)
19. Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for Object-Oriented design complexity: Implications for software defects. IEEE Transactions on Software Engineering 29(4), 297–310 (2003)
20. Tayi, G.K., Ballou, D.P.: Examining data quality. Commun. ACM 41, 54–57 (February 1998)
21. Van Engelen, R.A., Gallivan, K.A.: The gsoap toolkit for web services and peer-to-peer computing networks. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02). pp. 128–135. IEEE Computer Society, Washington, DC, USA (2002)