Managing Web Service Interface Complexity via an OO Metric-based Early Approach

Cristian Mateos, **Alejandro Zunino**
ISISTAN Research Institute, UNICEN University
{cristian.mateos, alejandro.zunino}@isistan.unicen.edu.ar
Sanjay Misra
Covenant University, Nigeria
ssopam@gmail.com
Diego Anabalon, and **Andres Flores**
GIISCO Research Group, National University of Comahue
{diego.anabalon, andres.flores}@fi.uncoma.edu.ar

Abstract

Web Services have been steadily gaining maturity as their adoption in the software industry grew. Accordingly, metric suites for assessing different quality attributes of Web Service artifacts have been proposed recently – e.g., for services interfaces in WSDL (Web Service Description Language). Like any other software artifact, WSDL documents have several inherent attributes (e.g., size or complexity) that can be measured. We present an approach to prevent a high complexity on services interfaces (WSDLs), to ease consumers to reason about services' offered functionality. Mostly, WSDLs are automatically derived from object-oriented (OO) source code, with a likely impact on complexity. Thereby, we study the statistical relationships between a recent metric suite of service interface complexity (proposed by Baski & Misra) and the well-known Chidamber & Kemerer's OO metric suite (applied to service implementations), on a data-set of 154 real-world services. First, a theoretical validation of Baski & Misra's suite (using Weyuker's properties) is presented, to prove the ability to measure complexity in WSDL documents. Then, after finding high correlation between both metric suites, we have conducted a series of experiments to analyze how certain refactorings on services' source codes prior to derive WSDLs might reduce complexity. In this way, our approach exploits OO metrics as development-time indicators, to guide software developers towards obtaining less complex service interfaces.

Keywords: Web Services, Code-First, Web Service Interface Complexity, Weyuker's Properties, Object-oriented Metrics

1 Introduction

Service-Oriented Computing (SOC) is a paradigm that allows developers to build new applications by composing loosely coupled pieces of existing software called *services*. Services are provided by third parties, the *providers*, who only expose service interfaces to the outer world and hide technological and implementation details. By means of these interfaces, potential *consumers* can determine what a service offers from a functional perspective and remotely invoke it from their applications. In this way, SOC promotes service reuse, and thus speeds up client application development.

Nowadays, the high availability of broadband and ubiquitous connections allows users to reach the Internet from everywhere and at every time, enabling in turn the invocation of network-accessible services from new applications. In fact, the number of published services has increased in the last few years, which has generated a global scale marketplace of services where providers offer their services interfaces using communities of registries [1]. Due to this, services are often implemented using standard Web-enabled languages and protocols, and thus are called *Web Services*.

Much of the success of an individual Web Service depends on the quality of its interface, because this is the only artifact consumers have access to when reasoning about the offered functionality [2]. Even when many approaches that simplify the task of finding appropriate services exists [3], it is the user who has the final word on which service is selected from a list of potential candidates. This selection unavoidably requires to inspect the WSDL documents of the candidate services.

Web Service interfaces are specified using WSDL, an XML-based format for describing a service as a set of operations, which can be invoked via message exchange. Like any other software artifact, a WSDL document has attributes, all of which can be measured [4]. In this line, previous research works have studied how to assess Web Service quality from their WSDL interfaces [5, 4, 6]. Particularly, [5] studies a catalog of common bad practices in WSDL specification affecting service discoverability, whereas [6] proposes a suite of metrics to assess the complexity of Web Services interfaces. Based on these catalogs, service developers can modify and improve their WSDL documents until desired metrics values and therefore certain quality levels are met.

In most development scenarios, however, providers do not have full control over WSDL documents [7, 2]. This is since documents are not built by hand but they are automatically generated from services inner implementations. This approach is called *code-first* Web Service development. Particularly, in Java, this is done using *code-first tools* such as Java2WSDL [8], Java2WS [9], EasyWSDL [10], and WSProvide [11]. One advantage of using these tools is that WSDL specification is automatic, and thus expertise in WSDL itself is not required. However, having no control of WSDL specifications might result in WSDL documents that suffer from quality problems [7] as measured by the above metrics.

Thereby, our aim is to investigate the hypothesis that providers can indirectly improve interfaces according to these WSDL-level metrics, by following certain programming guidelines at service implementation time – i.e., prior to automatic WSDL generation. In this sense, previously [7] we intended to explain the root causes of discoverability problems associated to WSDL specifications, by performing a statistical correlation analysis between service implementation metrics and interface discoverability anti-patterns occurrences. In this paper, we study the feasibility of obtaining less complex WSDL documents by using two metrics suites: a metric suite for Object-Oriented (OO) source code (onto services implementation), and the metric suite for WSDL complexity by Baski & Misra [6]. Similarly to the approach in [12], in which a model predicts Web Service performance, our approach exploits OO metrics as development-time indicators, to guide software developers towards obtaining less complex service interfaces.

We found a statistically significant correlation between (source code-level) OO metrics and the catalog of (WSDL-level) service metrics by Baski & Misra [6]. To the best of our knowledge, this is to date the only catalog of metrics focused on measuring service complexity and data-types from WSDL interfaces. A corollary of this finding is that providers could consider applying simple early code refactorings to avoid obtaining complex services upon generating service descriptions. By "early" we mean using OO metrics from the code implementing services to indirectly control the levels of "future" complexity in generated WSDL documents. This paper illustrates how well-known refactorings, particularly those from Martin Fowler's catalog [13], could be used. Although our findings do not depend on the programming language in which Web Services are implemented, we focus on Java, which is widely used in back-end development. To evaluate our approach, we have done experiments with a data-set of 154 real Web Services, and four of the most popular source-to-WSDL mapping tools for Java.

It is worth noting that this paper reports results that are part of a bigger research project of ours aiming at studying the relationships between traditional OO metrics, and the existing WSDL catalogs of discoverability metrics [7] and complexity metrics [14]. In this line, this paper extends [15], which reports a preliminary correlation analysis between Chidamber & Kemerer's metrics [16] and Baski & Misra's metrics – based on WSDL documents obtained via Axis Java2WSDL [8] – by including two major additions:

- A significantly extended analysis using four code-first tools for translating Web Service code to WSDL apart from Axis Java2WSDL, namely OW2 EasyWSDL [10], CXF Java2WS [9] and JBoss WSProvide [11]. This will bring the findings to more code-first Web Service developers within the Java community. To make the paper self contained, we have included some of the explanations from [14], particularly definitions and examples of the target WSDL complexity metrics.
- Based on the former analysis, we applied refactoring operations on service implementations to alter the values of OO metrics and indirectly the values of WSDL-level complexity metrics. In this way, we quantify the impact of refactorings on services interface complexity as measured by Baski & Misra.

Section 2 reviews related work focused on quality metrics for services interfaces and service interface improvement. The Section also explains the metrics by Baski & Misra. A theoretical validation of these metrics, and an overview of the statistical hypotheses underpinning our metric-based early approach to address Web Service interface complexity are detailed in Section 3. Particularly, Subsection 3.2 presents the theoretical validation. Subsection 3.3 presents the statistical validation that evidence the correlation of OO metrics with the metrics proposed in [6]. Section 4 presents experiments applying refactoring operations that can be used to keep service interface complexity low. Outcomes of the study are described in Subsection 4.3. Conclusions and future work are presented afterwards.

2 Background and Related Work

WSDL allows providers to describe two main aspects of a service, namely what it does (i.e., its functionality) and how to invoke it (i.e., its binding-related information). Consumers use the former part to match external services against their needs, and the latter part to actually interact with the selected service. Service functionality is described as a *port-type* $W = \{O_0(I_0,R_0),..,O_N(I_N,R_N)\}$, which lists operations O_i that exchange input and/or 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 fragments between providers and consumers of services, and vice-versa. Data is represented by using data-type definitions in XML Schema Definition (XSD), a language to define the structure of an XML construct. XSD offers constructors for defining simple data-types, restrictions, and 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 different WSDL documents so as to achieve cross-document data-type reuse.

There have been different research efforts to measure quality –in its various forms– in WSDL-described services interfaces, but also to improve it, though in a comparative smaller quantity. There are also some works that apply an early approach to quality improvement for general software quality metrics. The next subsections summarize these works.

2.1 Quality Metrics for Service Interface Descriptions

Indeed, previous research has emphasized the importance of service interface quality. The work of Rodriguez et al. [5] identifies a suite of common bad practices or "anti-patterns" found in Web Service interfaces, which impact on the discoverability and readability of services. Discoverability refers to how accurately a service can be retrieved from a registry when a user issues a relevant query. Readability is the ability of a service interface description of being self-explanatory, i.e., no extra software artifact apart from a WSDL is needed to understand the functional purpose of a service.

The suite consists of eight bad practices. To assess the discoverability (and readability) of a WSDL document, one could account bad practices occurrences because the fewer the occurrences are, the better the WSDL document is. Then, the authors offer a tool called Anti-patterns Detector [5], which automatically computes the bad practices on an input WSDL document.

Sneed [4] describes a metric suite for WSDL documents, ranging from common size measurements like lines of code and number of statements, to metrics for measuring the complexity and other miscellaneous aspects. All the involved metrics can be computed from a service interface in WSDL, since the metric suite is purely based on WSDL schema elements occurrences. The most relevant complexity metrics included in the suite are Interface Data Complexity, Interface Relation Complexity, Interface Format Complexity, Interface Structure Complexity, Data Flow Complexity (Elshof's Metric), and Language Complexity (Halstead's Metric). The miscellaneous metrics are Modularity, Adaptability, Reusability, Testability, Portability, and Conformity.

2.1.1 Baski & Misra Metrics Suite

Baski & Misra [6] presents a metric suite ("BM suite" from now on) whose cornerstone is that the effort required to understand data flowing to and from the interfaces of a service can be characterized by the structures of the messages that are used for exchanging and conveying the data. The authors define four metrics, whose formulas are shown in Table 1: *Data Weight* (DW), *Distinct Message Ratio* (DMR), *Message Entropy* (ME) and *Message Repetition Scale* (MRS). These metrics can be computed from a service interface in WSDL, since the suite is purely based on WSDL and XSD schema elements occurrences. Below, we further explain these metrics since the study in this paper is based upon them.

Data Weight metric *(DW)* This metric, with Formula (1), provides a mean to compute the structural complexity of the data-types conveyed in service messages. In Formula (1), $C(m_i)$ broadly counts and weights the various XSD elements, XSD simple/complex data-types and XSD restrictions exchanged by the parts of message m_i [17]. According to the authors, $C(m)$ can be taken as an indicator of the effort required to understand a message m based on its XSD structure [17]. In WSDL, message parts (operation arguments) are associated with element definition/declarations or data-type definitions from an XSD. Since each element or data-type definition in the schema is assigned a weight value that reflects their inherent complexity degree, wp_j has the same weight value as the associated element or data-type definitions in the XSD. This is, wp_j is equals to we if the part references an element declaration in the schema, or wt if the part has a data-type reference to a schema element definition. Moreover, wt further depends on whether the data-type is simple or complex. See [17] for more details on how we and wt are determined.

Table 1: BM Metrics Suite

Metric	Formula
Data Weight *(DW)*	$DW(wsdl) = \sum_{i=1}^{n_m} C(m_i)$ (1) n_m: number of messages (input/output) of a WSDL. $C(m) = \sum_{j=1}^{\#parts(m)} wp_j$ wp_j: weight value of the j-th part of message m.
Distinct Message Ratio *(DMR)*	$DMR(wsdl) = \frac{DMC(wsdl)}{n_m}$ (2) n_m: total number of messages in the WSDL.
Message Entropy *(ME)*	$ME(wsdl) = \sum_{i=1}^{DMC(wsdl)} P(m_i) * (-log_2 P(m_i))$ (3) $P(m_i) = \frac{nom_i}{n_m}$ nom_i: number of occurrences of the i^{th} message, n_m: total number of messages in the WSDL.
Message Repetition Scale *(MRS)*	$MRS(wsdl) = \sum_{i=1}^{DMC(wsdl)} \frac{nom_i^2}{n_m}$ (4) nom_i: number of occurrences of the i^{th} message, n_m: total number of messages in the WSDL.

In summary, DW values are positive integers. The bigger the DW of a WSDL document is, the more dense the parts of its messages are. Then, DW values should be kept low.

Distinct Message Ratio metric *(DMR)* This metric, with Formula (2), complements DW by considering the case of WSDL documents having many messages with the same structure. As the number of similarly-structured messages in a WSDL increases, it is arguably less effort-demanding to reason about similarly-structured messages than structurally different ones as a result of gaining familiarity with repetitive messages while manually inspecting the WSDL [6].

The DMC (*Distinct Message Count*) function counts the number of distinct-structured messages in a WSDL. It is computed from the possible $[C(m_i), \#parts(m_i)]$ pairs, i.e., the complexity value and the total number of parts (either input or output operation arguments) that each message m_i contains [6]. To better illustrate the DMC function, Figure 1 shows two WSDL documents defining an operation to return the weather report in a city. For the sake of clarity, the WSDL code uses the RPC binding style, and thus individual message parts are specified separately instead of within a per-message XSD complex data-type (as opposed to the Document binding style).

The sample WSDL defines two messages for the operation (GetWeatherReportIn and GetWeatherReportOut) with simple data-types (city and report, respectively). Computing DMC would first lead to two pairs $[C(GetWeatherReportIn), 1]$ and $[C(GetWeatherReportOut), 1]$. Since both messages have the same XSD complexity according to $C(m)$, $C(GetWeatherReportIn) = C(GetWeatherReportOut)$. As a result, DMC outputs zero since there are not distinct pairs. On the other hand, by looking at the WSDL document on the bottom, GetWeatherReportIn has two arguments. The pairs are $[C(GetWeatherReportIn), 2]$ and $[C(GetWeatherReportOut), 1]$, resulting in $DMC = 2$. Then, the documents in Figure 1 have a DMR of $0/2 = 0$ (top) and $2/2 = 1$ (bottom), respectively.

Then, the DMR metric provides a number in the range $[0, 1]$, where 0 means that all defined messages are similarly-structured (lowest complexity from the point of view of the metric), and 1 means that all messages are dissimilar (highest possible complexity). Then, DMR values should be kept low.

Message Entropy metric *(ME)* This metric, with Formula (3), exploits the percentage of similarly-structured messages that occur within a given WSDL document. Compared with DMR, ME also assumes that repetition of the same messages makes a developer more familiar with the WSDL document, but ME bases on an alternative differentiation among WSDL documents in this respect. The ME metric has values in the range $[0, log_2(n_m)]$. A low ME value shows that the messages are consistent in structure, which means that the complexity of a WSDL document is lower than that of other WSDLs having equal DMR values [6]. Then, ME values should be kept low.

```
  ...
  <wsdl:types>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="report" type="xsd:string"/>
  </wsdl:types>
  <wsdl:message name="GetWeatherReportIn">
    <wsdl:part element="tns:city" name="cityIn"/>
  </wsdl:message>
  <wsdl:message name="GetWeatherReportOut">
    <wsdl:part element="tns:report" name="reportOut"/>
  </wsdl:message>
  ...
```

```
...
<wsdl:types>
  <xsd:element name="latitude" type="xsd:float"/>
  <xsd:element name="longitude" type="xsd:float"/>
  <xsd:element name="report" type="xsd:string"/>
</wsdl:types>
<wsdl:message name="GetWeatherReportIn">
  <wsdl:part element="tns:latitude" name="latitudeIn"/>
  <wsdl:part element="tns:longitude" name="longitudeIn"/>
</wsdl:message>
<wsdl:message name="GetWeatherReportOut">
  <wsdl:part element="tns:report" name="reportOut"/>
</wsdl:message>
...
```

Figure 1: DMC function: Examples.

Message Repetition Scale metric *(MRS)* This metric, with Formula (4), analyzes variety in structures of WSDL documents. MRS measures the consistency of messages by considering $[C(m_i), \#parts(m_i)]$ pairs in the given WSDL document. MRS values are in the range $[1, n_m]$. When comparing two or more WSDL documents, a higher MRS means less effort from the developer to reason about the messages structures due to the repetition of similarly-structured messages [6]. Then, MRS values should be kept high.

2.2 Approaches to improve Services Interfaces

Two main approaches to improve services interfaces exist in the literature. One approach occurs at the specification phase of services and WSDL documents. The other approach is called "early", since it deals with the improvement of interfaces during the implementation phase of the services, i.e., prior to obtain their corresponding WSDL documents.

The work of [5] fits in the first approach mentioned, since it identifies WSDL bad practices and supplies guidelines to remedy them. A requirement inherent to applying these guidelines is following *contract-first*, a method that encourages designers to first build the WSDL document of a service and then supply an implementation for it. In this context, the term "contract" means service interface. However, the most used method to build WSDL documents in the industry is *code-first* [2], which means that one first implements a service and then generates the corresponding WSDL document by automatically deriving the interface from the implemented service code. This means that WSDL documents are not directly created by humans but are instead automatically derived via language-dependent tools, which essentially map source code to WSDL code.

With regard to the *early* approach [7], the idea is to anticipate potential quality problems (in a broad sense) in service interfaces. Conceptually, the approach identifies refactoring operations for service implementations that help to avoid problems in service interfaces. The main hypothesis of this approach is the existence of statistical relationships between two groups of metrics, one at the service implementation level and another at the service interface level. This implies that at least one metric in the former group could be somehow "controlled" by software engineers attempting to obtain better WSDL documents, with respect to what is measured by the metrics that belong to the latter group. This means that if a developer modifies his/her service implementation and in turn this produces a variation on implementation level metrics, this change will be propagated to service interface metric values, assuming that both groups of metrics are correlated. Thus, key to this approach is determining to what extent implementation level metrics relate to interface level ones.

The work presented in [7] studies the relationships between service implementation metrics and the occurrences of the bad practices investigated in [5], when such interfaces are built using the code-first method. This bad practices come with a number of metrics not to assess complexity but to measure service discoverability, i.e., how effective is the process of ranking or locating an individual service based on a meaningful user query that describes the desired functionality. In [7], the authors gathered seven classic OO metrics from several service implementations, namely

Chidamber & Kemerer's [16] CBO (Coupling Between Objects), LCOM (Lack of Cohesion of Methods), 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 [18] and the well-known LOC (Lines of Code) and TPC (Total Parameter Count) metrics. Additionally, they considered four ad-hoc metrics of their own. Then, the authors created a data-set of 90 publicly available code-first Web Services projects and analyzed the statistical relationships among metrics. Finally, the correlation analysis showed that some WSDL bad practices that attempt against service discoverability are strongly correlated with some implementation metrics.

In this paper, alternatively, we propose to use OO metrics as early indicators to guide software developers towards obtaining less complex service interfaces, by correlating the seven OO metrics previously mentioned with the WSDL ones described in [6]. Moreover, this paper shows how well-known code refactorings, particularly that of Martin Fowler's catalog [13], can be used to early modify WSDL documents complexity. Therefore, unlike [7], in this work we are not concerned with the discoverability quality attribute associated to Web Service interfaces, but only with complexity.

2.3 Other early Approaches to Software Assurance

Indeed, the early approach to software quality assurance is not exclusive to Web Service development. Many works use OO metrics to predict and early control conventional software aspects with both a negative connotation (e.g., software defects), or positive ones (e.g., performance or potential popularity). Below we mention a few of such works (representatives in this line) to give the reader a big picture on the topic and better illustrate the concept of early code measurements.

The work in [19] employs OO metrics for preventing software defects, specifically those reported by customers when using a software and those identified during customer acceptance testing. The authors state five hypotheses associating one or more metrics with an increase in the number of defects. Here, the dependent variables are the defect count. To test the hypotheses, the authors manually collected metrics from an e-commerce suite developed in C++ and Java and compared them against defect resolution logs from a configuration management system. Empirical evidence supporting the role of OO design metrics in determining software defects was achieved by using a data-set comprising 706 C++ and Java classes.

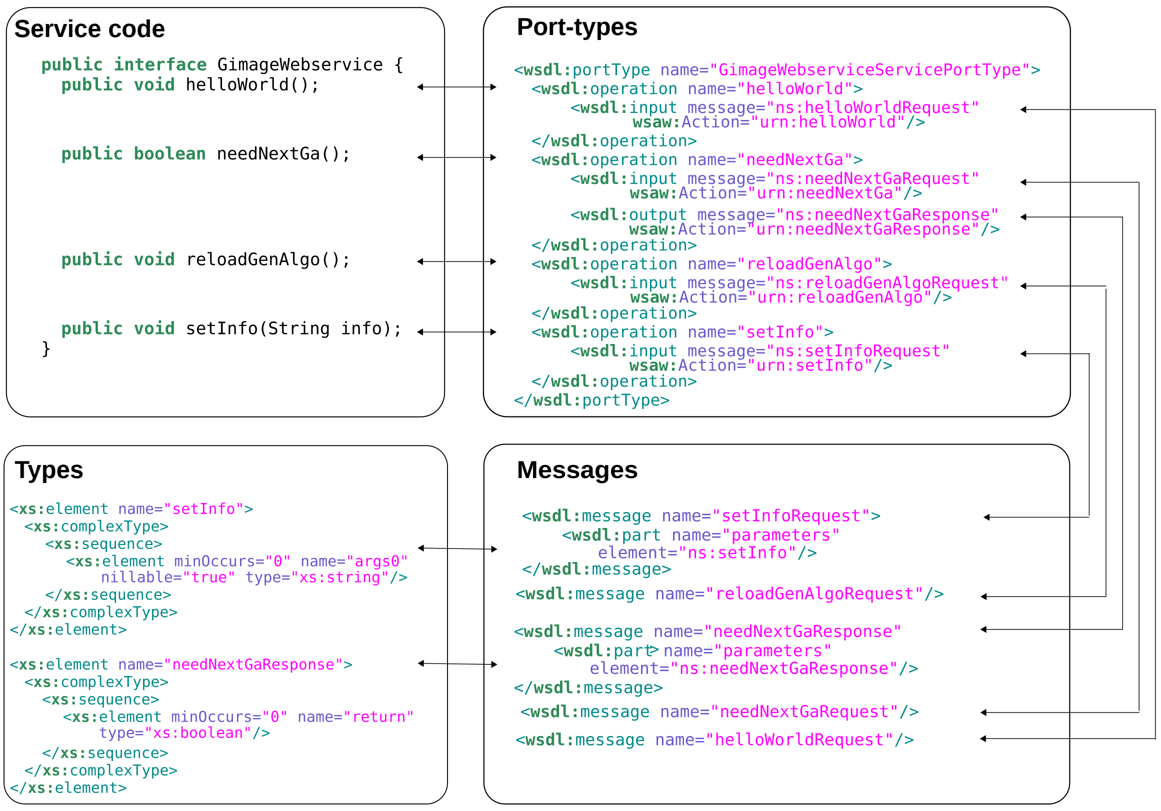
The correlation between software bugs and Chidamber & Kemerer's suite has been assessed for the well-known Mozilla project in [20]. They used 8,936 different bug entries that had been reported in the bug tracker used during the development of the project. On the other hand, the authors gathered OO metrics from 3,192 C++ classes as independent variables. For the analysis of relationships between bugs and OO metrics the authors performed a statistical correlation study, concluding that CBO seems to be the best in predicting bugs, but LCOM is the most practical since LCOM performed fairly well and it can be easily calculated. All in all, this works aligns with the idea that correlating OO metrics and software defects has proved to be a viable approach to bug detection [21].

Moreover, in [22] the authors evaluated the relations between OO metrics and the popularity of real open source projects. Popularity was quantified based on the number of downloads and members of each project. Furthermore, the authors found that CBO, LCOM and LOC and the total number of code modules were the OO metrics that statistically influenced the independent variables, i.e., downloads and members. Experiments were carried out by using a data-set of 6,773 projects implemented in C, which were extracted from SourceForge. In the end, the findings were that higher CBO implies less popularity, higher LOC suggests more functionality and maturity and hence more attractiveness, and more modules in a project seems to attract more members.

3 Addressing Service Interface complexity: An early Approach

A fact supporting our work is that, in practice, software developers first implement a service and then generate the corresponding service interface, by automatically deriving the WSDL interface from the implemented code [2] by using programming language-dependent tools. This WSDL document building method is called *code-first*, and is the most popular development approach in industry. Moreover, such tools perform a mapping T, formally $T : C \to W$. 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 a return message R, while each message comprises an XSD data-type representing the parameters of the corresponding class method. Tools such as WSDL.exe, Java2WSDL and gSOAP rely on a mapping T for generating WSDL documents from C#, Java and C++, respectively.

Figure 2 shows the generation of a WSDL document using Java2WSDL. The mapping T in this case associated each public method from the example service code with an *operation* containing two *messages* in the WSDL document. Messages in turn have been associated an XSD data-type containing the parameters of that operation. Depending on the tool used some minor differences between the generated WSDL documents might arise [7]. For instance, for the



```
public interface GimageWebservice {
  public void helloWorld();

  public boolean needNextGa();

  public void reloadGenAlgo();

  public void setInfo(String info);
}
```

```
<wsdl:portType name="GimageWebserviceServicePortType">
  <wsdl:operation name="helloWorld">
      <wsdl:input message="ns:helloWorldRequest"
             wsaw:Action="urn:helloWorld"/>
  </wsdl:operation>
  <wsdl:operation name="needNextGa">
      <wsdl:input message="ns:needNextGaRequest"
             wsaw:Action="urn:needNextGa"/>
      <wsdl:output message="ns:needNextGaResponse"
             wsaw:Action="urn:needNextGaResponse"/>
  </wsdl:operation>
  <wsdl:operation name="reloadGenAlgo">
      <wsdl:input message="ns:reloadGenAlgoRequest"
             wsaw:Action="urn:reloadGenAlgo"/>
  </wsdl:operation>
  <wsdl:operation name="setInfo">
      <wsdl:input message="ns:setInfoRequest"
             wsaw:Action="urn:setInfo"/>
  </wsdl:operation>
</wsdl:portType>
```

```
<xs:element name="setInfo">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="args0"
          nillable="true" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="needNextGaResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="return"
          type="xs:boolean"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
 <wsdl:message name="setInfoRequest">
     <wsdl:part name="parameters"
         element="ns:setInfo"/>
 </wsdl:message>
<wsdl:message name="reloadGenAlgoRequest"/>

<wsdl:message name="needNextGaResponse"
    <wsdl:part name="parameters"
        element="ns:needNextGaResponse"/>
</wsdl:message>
 <wsdl:message name="needNextGaRequest"/>
<wsdl:message name="helloWorldRequest"/>
```

Figure 2: WSDL generation in Java through the Java2WSDL tool

same service, Java2WSDL generates just 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.

3.1 Methodology

In order to carry out our study we have settled the main goal, from which we formulated a hypothesis, and a process that includes certain theoretical and technological choices, as follows.

Goal. Preventing a high complexity on Web Services interfaces described by WSDL documents, in order to ease consumers to make a comprehensive reasoning about services' offered functionality.

Much of the success of an individual Web Service depends on the quality of its interface, because this is the only artifact consumers have access to when searching for a required functionality to be fulfilled [2]. In addition, the widely adopted *code-first* approach affects providers in their ability to control the quality level (e.g., complexity) that a WSDL exhibits [7]. A solution is to follow certain programming guidelines at service implementation time, before automatically generating WSDLs. This leads to formulate the hypothesis of our study.

Hypothesis. *"It is possible to modify the complexity of a service interface WSDL document according to the BM suite of WSDL-level metrics, by making certain refactorings in the OO source code of the service implementation prior to automatic WSDL generation".*

Process. For testing this hypothesis we have performed the following steps:

1. *BM suite Theoretical Validation*: explaining that WSDL-level metrics *DW*, *DMR*, *ME* and *MRS* of the BM suite, adhere to desirable theoretical properties for any software *complexity* metric (Section 3.2);
2. *Statistical Correlation Analysis*: setting forth sound hypotheses on that seven OO source code-level metrics (onto service implementation) – e.g., Chidamber & Kemerer's [16] – might influence the above service interface (WSDL-level) complexity metrics of the BM suite. Statistical correlation experiments are provided to test the hypotheses set (Section 3.3);

3. *Refactoring Experiments*: selecting and applying refactoring operations on service implementations (OO source code) driven by the correlations found. The intent is to assess whether applying the refactorings on service implementations actually produces service interfaces with different complexities as measured by the WSDL-level metrics of the BM suite (Section 4).

Theoretical and Practical support. To carry out the process above, certain theoretical and practical-technological choices were made, as explained below.

- To perform step 1), we used the well-known Weyuker's validation framework [23] for software complexity metrics, which includes a formal set of nine properties. In short, the idea is that a good complexity metric should satisfy most of these properties [23].
- To support steps 2) and 3), we used a data-set of real world Web Service codes collected from open source project repositories. Concretely, we used a data-set of 154 services, which was the newest version available of the data-set described in [7] at the time of writing this article. This data-set of code-first Web Services is the first of its kind. All projects are written in Java, and were collected via source code search engines including Merobase, Exemplar and Google Code. All in all, the generated data-set provided the means to perform a realistic evaluation since the different Web Service implementations came from real-life software developers. Notice that this data-set has been gather mainly with static (design-time) experimental purposes. Hence, no executable versions were deployed to enable publishing in a public-accessable repository. Besides, the data-set is freely available to interesting readers, for whom we encourage them to contact the authors.

  To consider code-first tool incidence, we generated three WSDL data-sets by using Java2WSDL, EasyWSDL and WSProvide from the same initial Web Service data-set. We found that Java2WS generated almost exactly the same documents as WSProvide, but by embedding the XSD types in the WSDL documents. Since the BM suite is not affected by the place in which XSD types are declared (i.e., embedded as in Java2WS or in a separate file as in WSProvide), we left Java2WS out of the analysis. Although having XSD types embedded in WSDL documents has been identified as a bad practice that compromise the chance of a service of being effectively discovered [2], we ignored this since the paper does not concern service discoverability.

3.2 Theoretical validation of the BM suite

Next, we describe a theoretical validation of the BM suite through Weyuker's properties [23], a set of properties designed to evaluate software *complexity* measures, which has been historically a point of reference. Since the metrics operate at the WSDL level, below P and Q refer to WSDL documents.

Property 1: Non-coarseness. This property states that $(\exists P)(\exists Q)(|P| \neq |Q|)$, i.e., there are at least two WSDL documents with different metric values. Recall that DW (Data Weight) depends on the message complexity $C(m)$, which in turn increases as the number of parts of a message m increases. Therefore, assuming P being a WSDL with an arbitrary number of operations, one can always derive a new WSDL Q by adding a single part of arbitrary complexity to any message of P, and thus $DW(P) \neq DW(Q)$, because of the higher message complexity.

In addition, assuming that all messages in P have the same complexity and the same number of parts, then the value for DMC (Distinct Message Count) is $DMC(P) = 0$. Then, deriving Q from P by adding another message of different complexity or number of parts makes $DMC(Q) > 0$, and hence ME (Message Entropy) and MRS (Message Repetition Scale) of both documents will differ. From the two example WSDLs in Section 2.1.1, it can be seen that DMR (Distinct Message Ratio) also satisfies this property.

Property 2: Granularity. Let c be a non-negative number. Then, there are only finitely many programs of complexity c. All WSDL documents consist of a finite number of operations. Each operation consists of a finite number of messages and message parts. Since the BM suite depend on the structures of messages and their parts, there is a finite number of WSDLs with metric value c. Hence, all the metrics satisfy this property.

Property 3: Non-uniqueness. There are distinct WSDL documents P and Q such that $|P| = |Q|$, which means they have the same metric value. In WSDL, operations are composed of input and/or output messages (i.e., operation arguments and results, respectively). In this sense, if P is a one-operation WSDL with an input message of complexity i and an output message of complexity o, and Q is a one-operation WSDL with an input message of complexity o and an output message of complexity i (inverted input and output), then it is easy to see that the property holds for all metrics. First, $DW(P) = i + o = o + i = DW(Q)$. Second, since P and Q have one operation, their DMC function is zero, an hence DMR, ME and MRS are zero as well.

Property 4: Design implication. This property states that $(\exists P)(\exists Q)(P \equiv Q and |P| \neq |Q|)$, i.e., if two developers specify WSDL service interfaces for the same service functionality, the metric values might not be the same. Basically, all metrics satisfy this property on the grounds of the modeling features of XSD. One developer might choose to represent data in a specific message (argument) via simple data-types while another could use complex ones, and thus $DW(P) \neq DW(Q)$. Moreover, even with the same number of operations and messages, by using XSD complex data-types for message parts developers could establish different modeling criteria (e.g., using different hierarchy levels, defining restrictions or not for inner simple data-types, and so on), which impacts on message complexity in different ways. Then, $DMC(P) \neq DMC(Q)$, which makes DMR, ME and MRS values to differ as well.

Property 5: Monotonicity. Formally, the property states that $(\forall P)(\forall Q)(|P| \leq |P;Q| and |Q| \leq |P;Q|)$, which means that if two WSDL documents are concatenated into a new document $R = P;Q$, metric values for R must be at least equal to metric values for both P and Q. The common way of combining two WSDL documents is by concatenating their elements and prefix them using namespaces.

Since any WSDL document has at least one operation, and DW simply accumulates the complexities of messages, it follows that $DW(R) \geq DW(P)$ and $DW(R) \geq DW(Q)$. The case when $DW(R) = DW(P) = DW(Q)$ is when both P and Q has empty messages, i.e., messages without parts. Moreover, since R has more messages than both P and Q, even in the base case in which $DMC(R) = max(DMC(P), DMC(Q))$, the factor that depends on the number of messages in Formula (3) – i.e., $P(m_i)$ – remains the same. In addition, according to Formula (4), this factor increases quadratically with the number of similarly-structured messages in R.

DMR, on the other hand, does not satisfy the property. A counterexample would be P and Q each having two messages of complexities c_1 and c_2 such that $c_1 \neq c_2$. With this in mind, $DMR(P) = DMR(Q) = \frac{2}{2} = 1$, but $DMR(R) = \frac{2}{(2+2)} = 0.5 < 1$.

Property 6: Non-equivalence of interaction. This property has two associated simultaneous formulations, namely:

1. $(\exists P)(\exists Q)(\exists R)(|P| = |Q| and |P;R| \neq |Q;R|)$
2. $(\exists P)(\exists Q)(\exists R)(|P| = |Q| and |R;P| \neq |R;Q|)$

This property asserts that it is possible to find two WSDL documents of equal DW, DMR, ME or MRS values, which when separately concatenated into a third WSDL document results in different DW, DMR, ME and MRS values. Since concatenation order does not affect these metrics, we focus on formula #1.

First, having $DW(P) = DW(Q)$, and since DW simply accumulates the complexity of all messages in a WSDL, it follows that $DW(P;R) = DW(P) + DW(R) = DW(Q) + DW(R) = DW(Q;R)$, and thus the property is not satisfied. On the other hand, since DMR, ME and MRS depend on the DMC function, assuming two WSDL documents such that $DMC(P) = DMC(Q)$ and messages with different complexities, it is always possible to build R so that the complexity of its messages is equal to that of P and not Q. Then, $DMC(P;R) < DMC(Q;R)$ and therefore all the derived metrics are affected differently.

Property 7: Significance of permutation. There are documents P and Q such that this latter is formed by permuting the order of element declarations in P, and $|P| \neq |Q|$. Indeed, one can always build Q by flattening or deepening the structure of XSD data-types in P, and thus $C(m)$ is affected and in turn all metrics are affected.

Property 8: No change on renaming. The property simply states that if Q is a renaming of P, then $|P| = |Q|$. Indeed, since DW, DMR, ME and MRS are based on the internal structure of messages in a WSDL, changing the name of the WSDL does not affect its associated metric values.

Property 9: Interaction complexity. This property states that the complexity of two interacting WSDL documents will be greater than the sum of the metrics of the individual documents, or formally $(\exists P)(\exists Q)(|P| + |Q| < |P;Q|)$. As a corollary of the examples presented for Property 5, after concatenating two WSDL documents it is clear that there is no metric from the suite satisfying the property. However, the relaxed form of the Property 9, i.e., $(\exists P)(\exists Q)(|P| + |Q| \leq |P;Q|)$, is satisfied by all the metrics. This is also valuable when evaluating complexity metrics [24].

Summary. Table 2 summarizes the properties satisfied by the BM suite. To better illustrate the level of compliance of the suite with the properties, classical complexity metrics for procedural programs are also included: LOC (Lines of Code), Halstead's complexity [25] and McCabe's cyclomatic complexity [26]. Note that the metrics in the BM suite satisfy most of the properties, which qualify them as good complexity measures [23]. In particular, for the last property, a relaxed form is satisfied by the BM suite – i.e., denoted with '$*$' in Table 2.

Table 2: BM metrics suite: Satisfied Weyuker properties

Property	DW	DMR	ME	MRS	LOC	Halstead	McCabe
1	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2	Yes	Yes	Yes	Yes	Yes	Yes	No
3	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4	Yes	Yes	Yes	Yes	Yes	Yes	Yes
5	Yes	No	Yes	Yes	Yes	No	Yes
6	No	Yes	Yes	Yes	No	Yes	No
7	Yes	Yes	Yes	Yes	No	No	No
8	Yes	Yes	Yes	Yes	Yes	Yes	Yes
9	Yes*	Yes*	Yes*	Yes*	No	Yes	No
Total	8/9	8/9	9/9	9/9	6/9	7/9	5/9

(*) means that the relaxed form of the property is satisfied.

3.3 Statistical Correlation Analysis

We have selected seven well-known OO metrics (from our previous work [7]), namely Chidamber & Kemerer's [16] CBO (Coupling Between Objects), LCOM (Lack of Cohesion of Methods), WMC (Weighted Methods Per Class), RFC (Response for Class), plus the CAM (Cohesion Among Methods of Class) metric from Bansiya & Davis [18] and the well-known LOC (Lines of Code) metric and TPC (Total Parameter Count). Then we established seven hypotheses to test the statistical correlation among each of these OO metrics w.r.t. the complexity metrics of the BM suite. Below we formulate the hypotheses in a general manner but with a concrete explanation of its rationale.

Hypothesis 1 (H_1) The higher/lower the number of methods belonging to the class implementing a service (WMC metric, Weighted Methods per Class), the more different the complexity of its associated WSDL. The WMC metric [16] counts the methods of a class. First, since T-based code-first tools map each method onto an operation, more methods means more WSDL messages and parts, increasing the chance of much higher accumulated complexity and hence higher DW (Data Weight). Moreover, having more messages increases the probability that any pair of them are distinct in terms of the DMC (Distinct Message Count) function, and hence DMR (Distinct Message Ratio), ME (Message Entropy) and MRS (Message Repetition Scale).

Hypothesis 2 (H_2) The higher/lower the number of classes referenced from the class implementing a service (CBO metric, Coupling Between Objects), the more different the complexity of its associated WSDL. Basically, CBO [16] counts how many methods or instance variables defined by other classes are accessed by a given class. Code-first tools based on a mapping T include in resulting WSDL documents as many XSD definitions as objects are exchanged by service classes methods. We believe that increasing the number of external objects that are accessed by service classes may increase the likelihood of complex data-types definitions within WSDL documents, which affect messages complexity $C(m)$ and hence the BM suite.

Hypothesis 3 (H_3) The weaker/stronger the "tangling" among methods exposed by the class implementing a service (RFC metric, Response For Class), the more different the complexity of its associated WSDL. The RFC [16] metric measures the degree to which the methods in a class depend on other methods in the same class to accomplish their task. Then, low (or high) RFC means low (or high) independence between the methods, and hence the probability of exchanging similar data-types due to method call chaining might be different. This also affects method signatures, and potentially when converted to WSDL the BM suite metrics.

Hypothesis 4 (H_4) The higher/lower the cohesion among the methods in the class implementing a service (LCOM metric, Lack of Cohesion in Methods), the more different the complexity of its associated WSDL. The LCOM metric [16] attempts to measure whether a class represents a single abstraction or multiple abstractions. Then, if a class represents more than one abstraction, the chance that it implements many methods and defines several domain objects with different structure increases. Therefore, the probability that the resulting WSDL is more complex is lower or higher.

Hypothesis 5 (H_5) The more the number of code lines in the class implementing a service (LOC metric, Lines of Code), the more different the complexity of its associated WSDL. The rationale here is straightforward: more code lines in a class not only imply longer methods, but also more methods and in turn more arguments and return data-types, increasing the probability of affecting the complexity of the WSDL.

Hypothesis 6 (H_6) The higher/lower the cohesion among the methods of the class implementing a service as determined by the relatedness of the data-types of its method arguments (CAM metric, Cohesion Among Methods of Class), the more different the complexity of its associated WSDL. The CAM [18] metric computes the relatedness among methods of a class based on the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods. Then, according to this definition, CAM might influence the complexity of the service at the WSDL level.

Hypothesis 7 (H_7) The higher/lower the total number of parameters among the methods of the class implementing a service (TPC metric, Total Parameter Count), the more different the complexity of its associated WSDL. The rationale of this hypothesis is similar to that of H_1, but instead of considering the number of methods, the number of parameters is taken into account.

3.3.1 Spearman Correlation

To test the above hypotheses, we used the Spearman's rank coefficient to analyze whether the OO metrics related to service implementations are correlated with metrics from their generated WSDL documents or not. Broadly, the experiment consisted on gathering popular 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.

Regarding the correlation model, similarly to [7], the independent variables were the WMC, CBO, RFC, LCOM, LOC and TPC metrics. On the other hand, the dependent variables were the DW, 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. Therefore, the OO metrics have been automatically gathered by using an extended version of the *ckjm* [27] tool, whereas a software library[1] of our own was used to recollect the WSDL metric.

Table 3: BM suite metrics and OO metrics: Correlation

Code-first tools	**BM metrics**	*WMC* (H_1)	*CBO* (H_2)	*RFC* (H_3)	*LCOM* (H_4)	*LOC* (H_5)	*CAM* (H_6)	*TPC* (H_7)
Java2WSDL	*DW*	0.47	**0.79**	0.47	0.47	0.47	-0.48	0.57
	DMR	**-0.71**	0.21	**-0.71**	**-0.71**	**-0.71**	0.44	-0.4
	ME	**0.71**	0.6	**0.71**	**0.71**	**0.71**	**-0.78**	**0.66**
	MRS	**0.8**	-0.12	**0.8**	**0.8**	**0.8**	-0.56	0.49
EasyWSDL	*DW*	0.45	**0.74**	0.45	0.45	0.45	-0.48	0.56
	DMR	**-0.75**	0.16	**-0.75**	**-0.75**	**-0.75**	0.47	-0.43
	ME	**0.69**	0.53	**0.69**	**0.69**	**0.69**	**-0.8**	**0.7**
	MRS	**0.85**	-0.11	**0.85**	**0.85**	**0.85**	-0.61	0.52
WSProvide	*DW*	0.46	**0.78**	0.46	0.46	0.46	-0.49	0.53
	DMR	**-0.76**	0.18	**-0.76**	**-0.76**	**-0.76**	0.5	-0.4
	ME	**0.61**	0.6	**0.61**	**0.61**	**0.61**	**-0.73**	**0.71**
	MRS	**0.86**	-0.12	**0.86**	**0.86**	**0.86**	**-0.64**	0.5

Correlation Results Table 3 shows the found out relations between the OO metrics (independent variables) and the WSDL metrics (dependent variables). For denoting the coefficients that are statistically significant – i.e., those with its correlation factor above $|0.6|$ at the 5% level or $p - value < 0.05$ – we employed bold numbers and shaded cells in Table 3. The correlation coefficients are positive or negative (+, -). A positive relation means that when the

[1]http://code.google.com/p/wsdl-metrics-soc-course/

independent 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, dependent ones fall, and vice-versa. The absolute value, or correlation factor, indicates the intensiveness of the relation regardless of its sign.

It is known that a correlation between two variables does not imply causation. However, in principle these results are useful, as they imply that the independent variables (OO metrics) could be somehow controlled by software engineers attempting to obtain less complex WSDLs. However, as determining the optimal set of *controllable* independent variables would deserve an exhaustive analysis in another paper and depends on the data-set, we will focus on determining a minimalist sub-set of OO metrics and relations for the refactoring analysis in this paper using a reduction heuristic.

The heuristic relies on the fact that any pair of OO metrics could be also statistically correlated, thus controlling one or another would produce indistinct results. Based on a previous study [7] that shows the existence of groups of statistically dependent OO metrics, the heuristic selects one representative metric for each group.

Thereby, according to the results in Table 3, two groups of independent OO metrics can be observed. One group involves only the CBO (Coupling Between Objects) metric, which is the only OO metric strongly correlated to the DW complexity metric – i.e., with a factor of above $|0.6|$ for the three tools. The other group includes the rest of the OO metrics, which are strongly correlated to the remaining complexity metrics of the BM suite. Thus, by following the heuristic, one representative metric should be selected from this group. In this case, the WMC (Weighted Methods per Class) metric was selected, by being strongly correlated to the DMR, ME and MRS complexity metrics – i.e., with a factor of above $|0.6|$ for the three tools. Besides, the selection of WMC is due to a pragmatic aspect related to refactoring OO source code – as explained in the following section.

4 Experiments for Reducing WSDL Complexity

We have conducted two experiments to study the effect of refactoring service implementations, upon WSDLs complexity according to the BM suite. Based on the metric selection process described in the previous section, we tested if changes on services implementations – while decreasing CBO (Coupling Between Objects) and WMC (Weighted Methods per Class) values – reduces complexity on target WSDLs. Moreover, we selected from the Martin Fowler's catalog [13] two refactoring operations. On the one hand, the inverse of the REPLACE DATA VALUE WITH OBJECT [F-175] refactoring is applied to reduce the CBO metric. On the other hand, the MOVE METHOD [F-142] refactoring is applied to reduce the WMC metric. Note that the kind of code refactorings we target are those which actually might affect service public interfaces – i.e., parameter related, façade related. As explained in Section 3, a service interface is described as a *port-type* which lists operations and their input/output messages. Besides, code-first tools generate a WSDL document by a mapping involving the service implementation class representing the port-type, exposing as many operations as public methods the class defines. Hence, we leave out those refactorings which strictly deal with inner implementation details or private methods of the service code.

The selection of CBO and WMC metrics was explained in the previous section. In the case of CBO, the only strong correlation with the DW (Data Weight) complexity metric is evident. In the case of WMC, the choice from its group of OO metrics also considered a pragmatic side with popular IDEs that currently provide facilities to apply a refactoring operation like moving methods among different classes [7]. In addition, refactoring an OO source code based on other metrics in the same group like RFC (Response For Class) and LCOM (Lack of Cohesion in Methods) implies in practice to modify service method bodies, i.e., the inner functionality of the methods and not only the exposed methods signatures.

The two experiments are presented as follows. In Section 4.1 we studied the effect of refactorings based on both the CBO and WMC metrics. Each metric was firstly considered in a separated way and then in a combined way. In Section 4.2 we studied the effect of refactoring the WSDLs based only on the WMC metric. This time, specifics WMC values were considered as cut-off values. Finally, Section 4.3 discuses the outcomes of the experiments and their threats to validity.

4.1 Refactoring Experiment 1

In this experiment, we have studied the feasibility of reducing the BM suite metrics of WSDLs complexity by applying a reduction on the CBO and WMC OO metrics. This was performed in three rounds of refactorings of the service implementations. First, only the CBO metric was reduced. Second, only the WMC metric was reduced. Finally, both the CBO and WMC metrics were reduced in a combined way.

Applied Refactorings. In a first round of refactoring, we focused on reducing the CBO (Coupling Between Objects) metric from service codes. From the positive correlation between CBO and DW (Data Weight) follows that by reducing CBO, the DW metric is also reduced. We applied the inverse of REPLACE DATA VALUE WITH OBJECT [F-175]. The F-175 refactoring operation suggests to encapsulate data values, a.k.a. primitive types, into object classes. Then, the inverse of F-175 means to replace method arguments data-types with primitive types. For each service, we only modified the class that the WSDL generation tool receives as input, or the class offering service operations. For example, assuming that the class ZipCode has one primitive attribute of type long, and in turn the signature for a method of a service class is getTemperatureFor(ZipCode zipCode), then through the inverse of F-175 the signature would become getTemperatureFor(long zipCode). By repeating this refactoring operation on every method that exchanges an Object, the CBO of the class owning those methods is reduced. Once each service implementation class has been refactored, their target WSDL documents were automatically built using Java2WSDL, EasyWSDL and WSProvide. Then, we re-gathered the DW metric for the new WSDL documents.

It is worth noting that the performed first round of refactoring might lead to producing low quality code in *OO terms*. However, for our experimental goals this was the only way to test whether controlling metrics at the service implementation level might impact on target services interfaces. This is because the CBO metric represents the number of classes coupled to a given class, and this coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions. However, the three WSDL generation tools receive as input a Java interface, thus coupling only can occur through arguments declarations.

In a second round of refactoring, we took original service implementations classes that described service interfaces, and moved some of their methods into new interfaces. The goal was to reduce the WMC (Weighted Methods per Class) metric, since it is positively correlated to the ME (Message Entropy) metric. By applying the MOVE METHOD [F-142] refactoring operation on those classes having an excessive number of methods, a 50% decrement of the averaged WMC metric was achieved. In other words, we focused on reducing WMC by splitting the services having too many operations into two or more services so that on average the metric in the refactored services represented a 50% of the original value.

The results of the previous two rounds of refactoring evidence which operation refactoring results in less complex WSDL documents. This is important because in many cases not all refactoring operations might be executed. For instance, blindly replacing instances of domain classes with primitive types is not always a good programming practice to specify interfaces, as explained. However, on early development stages of services any refactoring operation might be feasible, and two or more refactoring operations might be sequentially applied. Thus, during a third round of refactoring, we refactored the implementations that resulted from the first round, but according to the second round. The goal of this refactoring round was to assess the effect of simultaneously controlling CBO and WMC on service interface complexity.

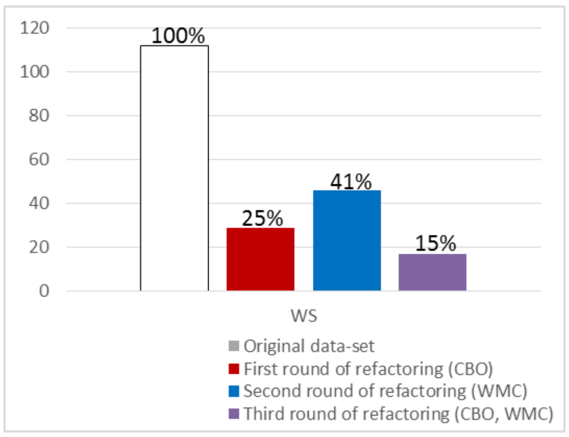
Regarding the third round of refactoring, by applying the refactoring operation associated with CBO and then the one associated with WMC, the effect was cumulative.

Results and Discussion. The implications of the employed refactoring operations on the DW (Data Weight) metric of the services interfaces are shown in Figure 3. Each data series represents a data-set of WSDL documents, being from left to right the data-set of original WSDL documents, the WSDL documents that resulted from applying the first round of refactoring, the WSDL documents of the second round of refactoring, and the WSDL documents for which we sequentially applied the first and second rounds. A column represents the averaged WSDL metric value gathered from the WSDL documents of each refactored data-set. Percentage values show the rate between a specific data-set value and the value gathered from the original data-set when using each tool.

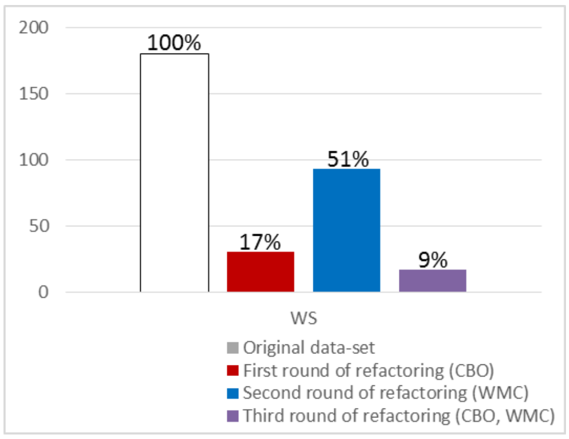
By replacing arguments that were classes with primitives from the interfaces of services implementations (Figure 3), the averaged data weight of WSDL documents falls to a 26% (Java2WSDL), 17% (EasyWSDL) and 23% (WSProvide) w.r.t. their original values. Specifically, this is shown by the second bar from the left. This decrement can be explained by revisiting the definition of the DW metric. DW is based on the complexity of a message $C(m)$. As explained in [6], $C(m)$ not only considers how many XSD elements are contained in within a data-type definition, but also the types of each independent XSD element. Thus, an XSD complex type, a restriction or a built-in type (such as xsd:string and xsd:int), are associated with different weights, being built-in types the least weighted elements. Therefore, by replacing Object arguments with primitive arguments, it is expected that a mapping T produces WSDL documents with lower data weights than before, as primitives are mapped onto built-in types by T.

Figure 3 also shows that though WMC was not strongly correlated with DW, it slightly impacted on averaged WSDL documents data weight, but not as much as with the first round of refactoring. Concretely, the DW metric falls only to 37-51% (depending on the tool used) on average, when applying a WMC-driven refactoring. In the Figure this can be seen in the third bar from the left. In the third round the averaged value achieved after sequentially applying both refactoring operations represented a 15% (Java2WSDL), 9% (EasyWSDL) and 13% (WSProvide) of the original average DW values. This is depicted in Figure 3 by the rightmost bar.

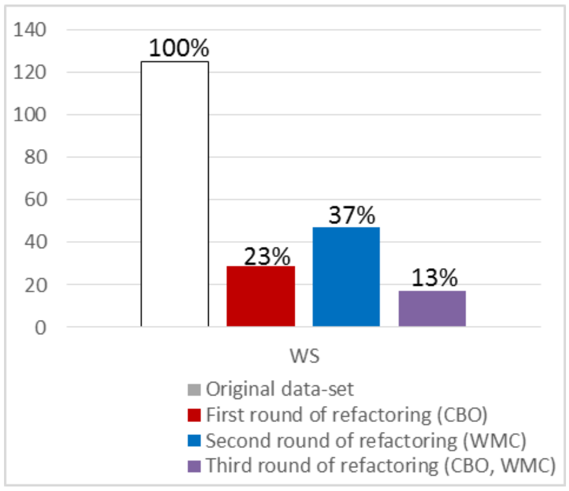
With respect to DMR (Distinct Message Ratio), ME (Message Entropy) and MRS (Message Repetition Scale) metrics, Figure 4 shows the implications of applying the three rounds of refactoring operations. Before analyzing the achieved results, recall that for the MRS metric, the higher the values, the less complex the WSDL documents. From Figure 4 it can be observed that the DMR, ME, and MRS metrics are simultaneously and best improved when only the refactoring associated to the CBO metric is applied. This stems from the fact that these metrics depend on $C(m)$



(a) Java2WSDL



(b) EasyWSDL



(c) WSProvide

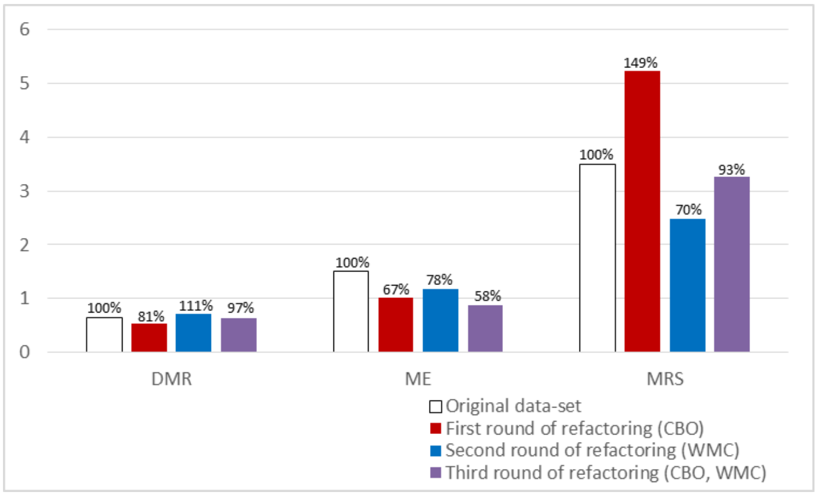
Figure 3: Impact on DW of refactoring operations driven by CBO and WMC metrics.

and n_m, i.e., the number of messages present in a WSDL document and the number of similarly-structured messages, respectively. First, DMR depends on $C(m)$, which as explained falls when arguments are primitives. Moreover, the impact on ME and MRS is explained by their dependency on the quantity of similarly-structured messages in a WSDL document. By flattening Object arguments into a finite set of primitive arguments, which are in turn mapped onto a finite set of similar messages, then an increment in the number of similar messages is produced, i.e., a positive impact on ME and MRS. On the other hand, the third round of refactoring appeared less efficient, but on the bright side it allows developers to also reduce DW. Thus, there is a trade-off between reduced complexity versus refactoring effort, which should be balanced by developers depending on their needs.

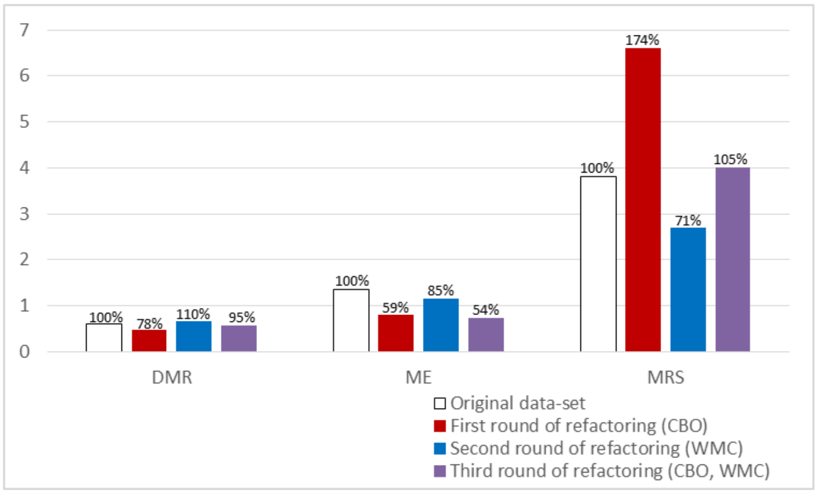
4.2 Refactoring Experiment 2

In this experiment, we have studied the feasibility of reducing WSDLs complexity by considering the effect produced particularly by the reduction of the WMC (Weighted Methods per Class) metric in various grades. As seen in Section 3.3, the WMC was strongly correlated with DMR (Distinct Message Ratio), ME (Message Entropy) and MRS (Message Repetition Scale) metrics for WSDL (Table 3). To perform the reductions, we have applied the MOVE METHOD [F-142] refactoring operation [13] on the service implementations. In addition, here we only focused on EasyWSDL as code-first tool to build the refactored WSDLs. In fact, the three code-first tools used along this work achieved similar results.

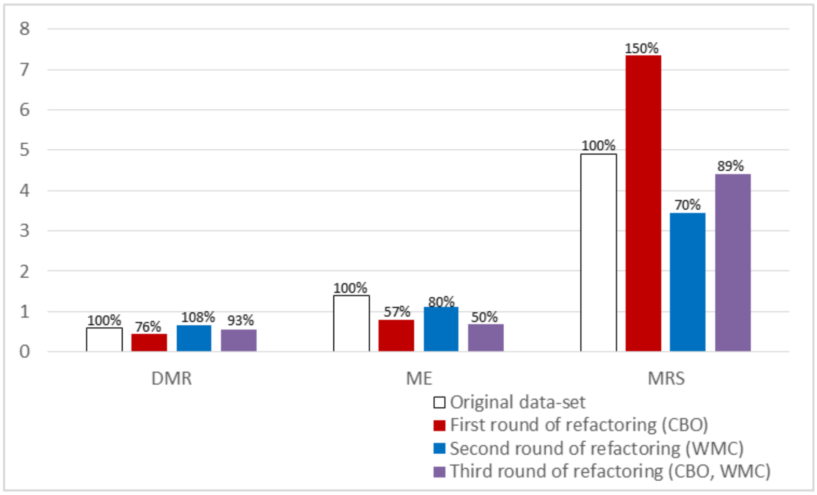
Applied Refactorings. In order to apply the refactorings based on the WMC metric, we have selected specific cut-off values: 1, 2, 5, 10 and 20 that respectively generate several new data-sets. This means, by choosing a cut-off value, the representative class implementing a service interface is fragmented to generate a new set of services that strictly achieve that WMC cut-off value. For example, if an original service consists of 13 operations, for a cut-off value of 5 ($wmc = 5$), then 2 services of 5 operations each are generated. In addition, for a cut-off value of 1 ($wmc = 1$) as



(a) Java2WSDL



(b) EasyWSDL



(c) WSProvide

Figure 4: Implications of refactoring operations driven by CBO and WMC metrics.

many new services as the total amount of operations contained in the original data-set, are generated. Figure 5 shows a different line for each new data-set generated, according to each WMC cut-off value. Thus, for each original service, the number of new services generated is shown, which were arranged in ascending order by the original WMC value.

As can be seen, as the cut-off value increases the number of generated services decreases. The highest line corresponds to $wmc = 1$, where each original service was fragmented according to its number of operations – 1

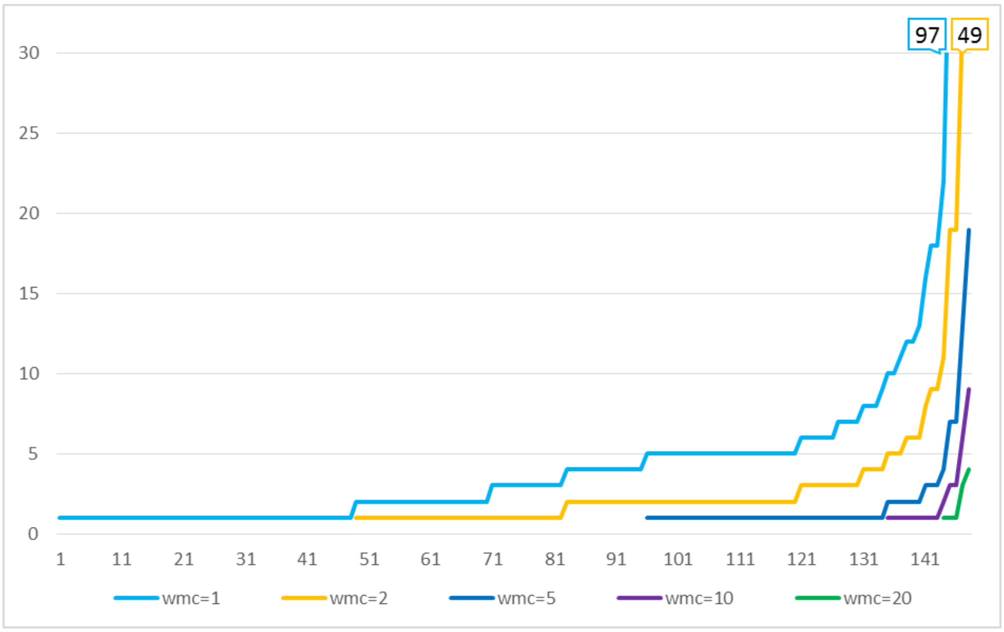


Figure 5: Generated data-sets for each WMC cut-off value.

operation per new service generated. The lowest line corresponds to $wmc = 20$, where each generated new service contains exactly 20 operations. Thus, in the case of the original service 148 (horizontal axis) of 97 operations, then 97 new services were generated for $wmc = 1$ and only 4 new services for $wmc = 20$. Each line begins in a service with at least the same WMC value as the cut-off representing the line. The original data-set includes few services with a large number of operations. Thereby, $wmc = 20$ line is the shortest. In addition, it is unlikely that a line of a greater WMC cut-off value may be above another line of lower WMC cut-off value in any point.

Effect on the WSDL-level metrics. Considering the way in which the new data-sets were generated, we have analyzed the variation of WSDL complexity metrics according to the WMC cut-off values. Figures 6, 7, 8 show the results of DMR, ME and MRS metrics respectively. For each original service we show the average metric value of its generated services, for each cut-off value.

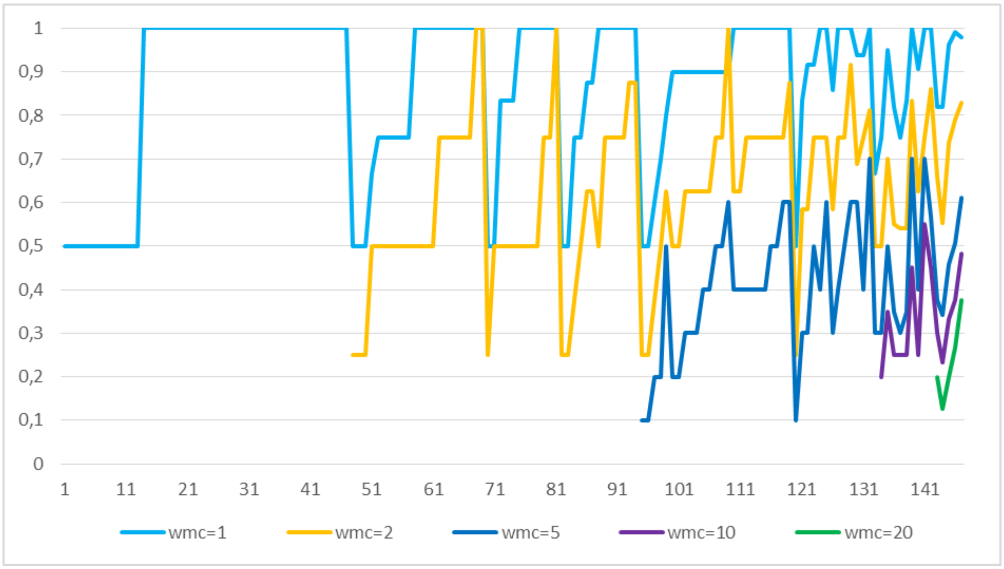


Figure 6: Impact on DMR of refactoring operations driven by different WMC cut-off values.

Regarding the DMR (Distinct Message Ratio) metric, from Figure 6 it can be observed the effect of complexity reduction as WMC increases its value. The $wmc = 1$ line ranges between 0.5 and 1 while the $wmc = 5$ line lower the range between 0.1 and 0.7. This fulfills the negative correlation between these two metrics – as seen in Table 3. For example, for the original service 144 (horizontal axis) of 39 operations, on $wmc = 1$ its DMR value was 0.82 while

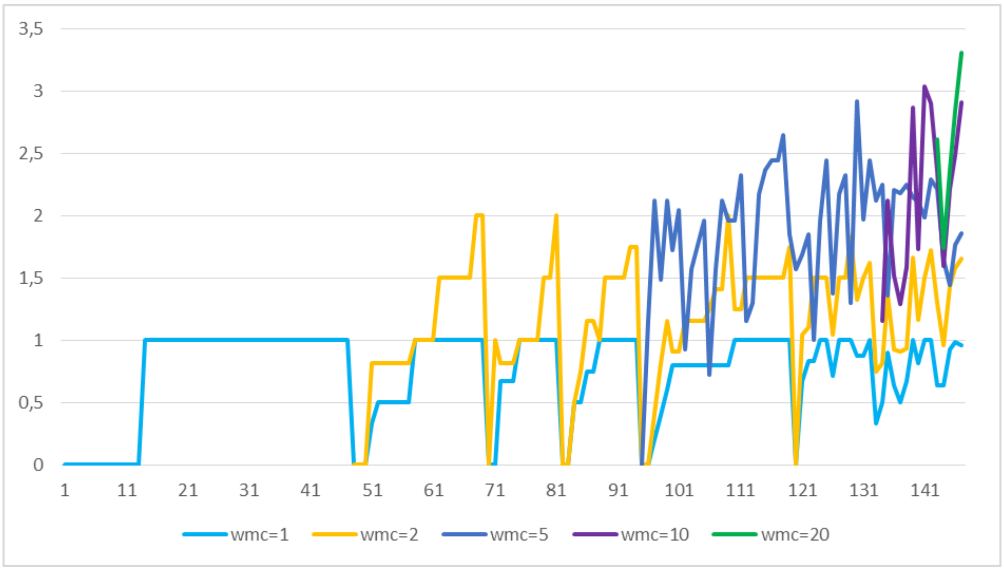


Figure 7: Impact on ME of refactoring operations driven by different WMC cut-off values.

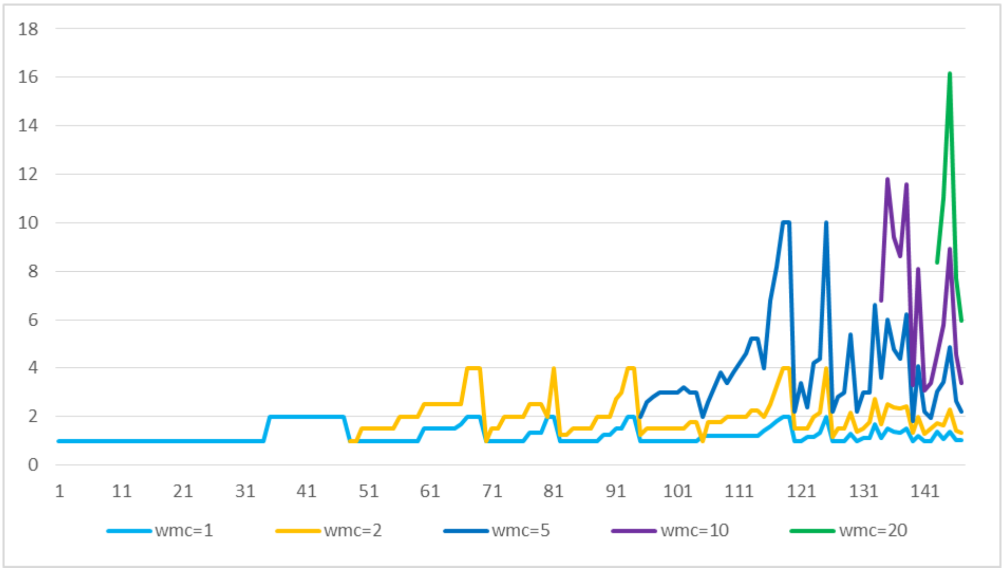


Figure 8: Impact on MRS of refactoring operations driven by different WMC cut-off values.

on $wmc = 20$ its value was reduced to 0.125. This means that, as the number of operations increases in a service, it is most likely that the number of similarly-structured messages increases as well. Hence, more familiarity with repetitive messages when reading a WSDL document is gained [6].

Regarding the ME (Message Entropy) metric, from Figure 7 it can be observed that the positive correlation with respect to WMC holds – as seen in Table 3. The $wmc = 1$ line ranges between 0 and 1 while the $wmc = 5$ line ranges mostly between 0.7 and 2.9. For example, for the original service 118 (horizontal axis) of 5 operations, on $wmc = 1$ its ME value was 1, on $wmc = 2$ its value was 1.5 and on $wmc = 5$ its value was 2.64. When services include only 1 operation its messages are more cohesive in structure, which eases the comprehension of the WSDL document. In contrast, when services include a large number of operations, it is likely that the messages could not be so structurally consistent. The upside is that having services with several operations means less WSDLs to index and search in service registries.

Regarding the MRS (Message Repetition Scale) metric, in Figure 8 the effect of complexity reduction as WMC increases its value is depicted. The $wmc = 1$ line ranges between 1 and 2 while the $wmc = 5$ line rises the range between 2 and 10. This fulfills the positive correlation between these two metrics – as seen in Table 3. For example, for the original service 119 (horizontal axis) of 5 operations, on $wmc = 1$ its MRS value was 2, on $wmc = 2$ its value was 4 and on $wmc = 5$ its value was 10. This explains that the same effect as in DMR applies to MRS concerning the familiarity of repetitive messages in a WSDL document [6].

Finally, to clearly observe the trend of metrics values in this experiment, we have calculated the average of the complexity metrics values on each new data-set. Figure 9 shows the DMR, ME and MRS metrics combined according to each WMC cut-off value. In this way, we can notice the effect of the complexity metrics, as WMC value increases or decreases.

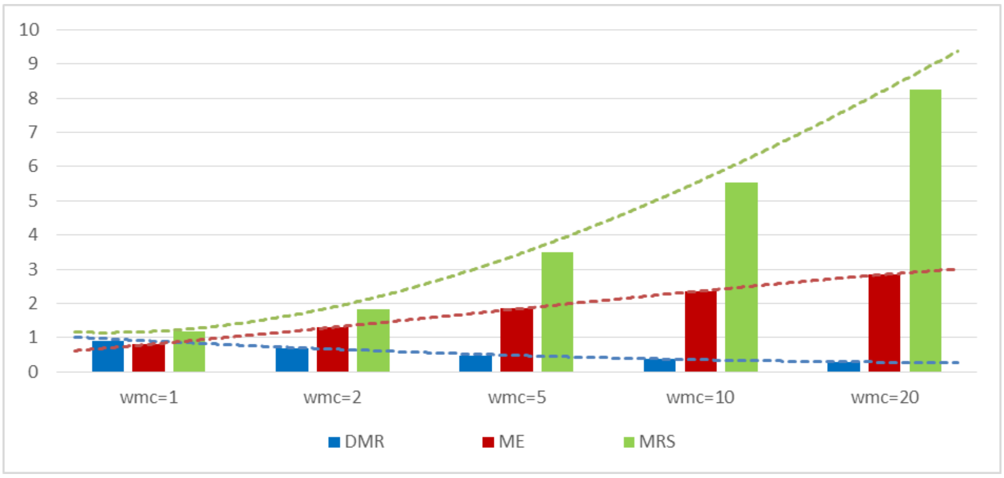


Figure 9: Average values of DMR, ME and MRS, by WMC cut-off values.

Each complexity metric is shown with its own value range. Regarding DMR, its value range is $[0,1]$. We may observe that on average this metric gets lower as WMC gets higher values. Besides, those values proportionally become lower without any hard fall. This is also seen by the smooth descending tendency line for the DMR metric. In case of ME, its value range is $[0, log_2(n_m)]$. We may observe that on average this metric proportionally increments its value as WMC gets higher. The ascending tendency line for ME metric, confirms this observation as well. Regarding MRS, its value range is $[1, n_m]$. Similarly to ME, this metric also proportionally rises as WMC values increase. This is also supported by the smooth ascending tendency line for the MRS metric.

Discussion. All in all, in this experiment we could confirm the strong correlation of the WSDL complexity metrics with respect to the WMC (Weighted Methods per Class) metric – as seen in Table 3. This means that the complexity of WSDL documents can be affected when the number of operations grows – i.e., at larger WMC cut-off values. Particularly, for DMR (Distinct Message Ratio) and MRS (Message Repetition Scale) the complexity is reduced for higher WMC values, since it is expected to keep DMR values low and MRS values high. In the case of ME (Message Entropy), its complexity is reduced for lower WMC values, since it is expected to keep ME values low.

To sum up, for lower values of WMC the ME metric complexity is better. However, in case of the DMR and MRS metrics the complexity worsen. One may come to expect that all the complexity metrics simultaneously obtain better values, which means the lowest complexity for WSDL documents. But, as seen in Figure 9 the results evidence that it is unlikely. On one side, only MRS and DMR metrics improve as WMC metric increases. This is due to both metrics find a higher number of similarly-structured messages upon a larger number of messages into a WSDL. On the other side, the ME metric achieves its better value when a larger number of messages are consistent in structure. However, when the number of messages increases, it is more likely to find the messages homogeneously distributed on each distinct structure.

Therefore, the solution to reduce WSDL complexity from the different angles cannot be only based on indirectly affecting the WMC metric. Thus, a developer should find a balance among the complexity metrics and particularly regarding WMC.

4.3 Discussions of Refactoring Experiments

From the results on both experiments and considering the correlations found between OO source code-level metrics and WDSL-level metrics (Section 3.3), we can analyze whether the general hypothesis of our study (see Section 3) has been confirmed.

The study was centered on how the complexity of WSDL documents affects comprehension for consuming services from target applications. Since a strong correlation was found between CBO (Coupling Between Objects) and WMC (Weighted Methods per Class) OO metrics w.r.t. the BM metric suite, we focused on making modifications at service source code to affect such OO metrics, and in turn to impact on the complexity of the corresponding WSDL document, as a *code-first* strategy. Modifications at service implementation were addressed by applying two refactoring operations from the Fowler's catalog [13]. The inverse of the REPLACE DATA VALUE WITH OBJECT [F-175] refactoring to affect the CBO metric, and the MOVE METHOD [F-142] refactoring to affect the WMC metric.

By focusing on the CBO-DW metrics strong correlation, complex data-types (being classes in a code) within messages were replaced by primitive data-types (Section 4.1). Despite that service interfaces become less complex, it is not always desirable to flatten object definitions, mostly because this might violate good data modeling principles. Consequently, Web Service developers should balance complexity and expressiveness of OO design for service operation parameters. The same applies in other software development contexts, e.g., whether database developers should denormalize a relational model for gaining performance or not. Here, deepest data normalization is the (theoretical) modeling principle to follow, while performance is the quality attribute that might force developers to disregard good modeling practices under certain circumstances. Then, refactoring –specially for DW– should not aim at achieving the lowest, but intermediate metric values.

Regarding the number of methods per service class, two refactoring strategies were applied concerning the WMC metric. The first refactoring (Section 4.1), showed that by reducing WMC to 50% on average in the whole data-set the WSDL document metrics presented lower values compared to the original data-set. Even when the obtained complexity levels were behind the ones achieved when CBO-driven refactorings were applied, the refactoring showed to be useful to control complexity. The second refactoring (Section 4.2), showed the tendency by each WMC cut-off value upon the DMR, ME and MRS metrics. A complexity reduction for DMR and MRS, was achieved for higher WMC values, while for ME the complexity is reduced for lower WMC values. Therefore, reducing the overall WSDLs complexity not only depends on the WMC metric. Thus, a developer should find a balance among the complexity metrics and particularly regarding WMC.

Threats to validity. There are two external validity threats. When studying the correlation between service implementation metrics and service interface ones, we have not considered the programming language as a factor. We have used a data-set comprising *Java-based* code-first services only. Nevertheless, considering that Java is very popular in back-end development and particularly Web Services [7], we ensure and deliver confident results at least to this large community. Besides, compared to [14], we have considered the factor given by the tool used for mapping from services implementation onto WSDL documents by using state-of-the-art Java tools. As this mapping directly affects resulting WSDLs, the used tool is more important than the language itself regarding validity. We will analyze the impact of the programming language, which will require building a code-first Web Service data-set for other languages.

Another threat concerns the size of the data-set, which includes 154 Web Services. From a purely software engineering standpoint, this is not a large sample. However, at present, this data-set is the largest code-first Web Service data-set for Java available in the literature [7].

Summary. A clear and strong influence is evident from OO metrics w.r.t. the BM suite, consequently the hypothesis of our study has been confirmed. Therefore, this early approach gives a pragmatic programming practice – as a code-first strategy to build Web Services – driven by OO metrics as development-time indicators. This fosters developers to be aware of implementation decisions, preventing a high complexity of services interfaces while enabling WSDL comprehension.

5 Conclusions

We presented an early approach to lowering service interface complexity, as computed by a relevant set of WSDL complexity metrics – i.e., the BM suite. This is achieved by performing refactoring operations on service implementations. Since the approach assumes that service interfaces are built using the *code-first* method, properly refactoring service implementations might positively impact on their associated WSDL documents.

Central to the proposed approach is a statistical correlation analysis showing that classic OO metrics obtained from service implementations are correlated to complexity metrics recollected from generated WSDL documents. By applying specific refactoring operations on service implementations, their corresponding WSDL documents become indirectly altered, presenting remarkable changes with regard to interface complexity. Refactorings were addressed by applying two operations from the Fowler's catalogue [13].

In the case of the CBO (Coupling Between Objects) OO-level metric, strongly related to the DW (Data Weight) WSLD-level metric, the inverse of the REPLACE DATA VALUE WITH OBJECT refactoring operation was applied. Message data-types were affected by turning object definitions into primitive data-types, reducing complexity at the expense of impoverishing expressiveness of a business domain data model. In the case of the WMC (Weighted Methods per Class) OO-level metric, strongly related to the DMR (Distinct Message Ratio), ME (Message Entropy) and MRS (Message Repetition Scale) WSDL-level metrics, the MOVE METHOD refactoring operation was applied. The number of WSDL operations – methods per service class – were affected generating new service interfaces with lower number of operations. As effect the number of data-type structures, associated to operations' messages, were also reduced. Consequently, the familiarity of repeated similarly-structured messages diminishes affecting comprehension of WSDL documents – i.e., worsen DMR and MRS metrics. However, the less the number of operations, the better the consistency of messages structures – i.e., few data-type structures largely repeated in similarly-structured messages

– increasing the ME metric. Therefore, reducing the overall WSDLs complexity requires developers being aware of design and implementation decisions, through balancing the OO-level metrics to avoid side effects related to a data model – e.g., less business domain expressiveness.

Currently, we are extending this study to deepen into the circumstances where the CBO metric can be affected by specific cut-off points – similarly to the second refactoring experiment for WMC metric. We expect to find the boundary line in which the BM metrics meet a balanced form. We are also working with another OO-level metric, the CAM (Cohesion Among Methods of Class) metric from Bansiya & Davis [18], which could be useful to identify proper fragmentation of services interfaces. In particular, the experiment with cut-off points for WMC disregarded the cohesion of generated WSDL documents (from an original WSDL). In this sense, other refactoring operations from the Fowler's catalog should also be applied – e.g., the MOVE CLASS refactoring operation. This could be a complement to the MOVE METHOD refactoring operation, after making a deep fragmentation. By analyzing operations of generated WSDLs to group them into clusters [28], a new improved service interface could be produced.

As future work, we consider to attend maintainability issues as well, for their important role on the providers side at keeping service quality levels. Hence, other refactoring operations could be selected to alter back-end classes of a service implementation. In this sense, we may base in [29, 30] where the Chidamber & Kemerer suite had been assessed as potential maintainability indicators of OO systems. Thereby, refactorings could be initially chosen upon their impact on OO metrics highly correlated to WSDL metrics – e.g., RFC (Response For Class) and LCOM (Lack of Cohesion in Methods). In addition, other recent OO metrics designed to measure code complexity (e.g., Class Complexity [31]) could be assessed on the correlation to the BM suite. Besides, the BM suite could be extended to adjust the existing metric definitions, or define complementary metrics. The new suite might consider two important aspects currently ignored, namely ambiguous and non-descriptive data-types names, which are known to make WSDL more complex in terms of readability [5]. In this sense, the new metrics could attempt to measure not only *structural* but also *textual/syntactic* complexity of XSD data-types, in line with existing efforts in the area [32]. Consequently, text-based refactoring actions should be identified and proposed. Another line of research includes using WSDL versioning techniques [33] to allow consumers that use old (unrefactored) WSDL documents versions to continue using the refactored WSDL documents until they migrate to the refactored versions. Finally, a current interesting research topic very close to our work relates to multi-objective optimization based on quality metrics [34, 35, 36]. Specifically, we intend to redirect our approach to exploit Search-Based Software Engineering techniques [37] to produce Web Services optimizing several metrics simultaneously. Opportunely, service providers could benefit from a programming practice driven by a multi-objective early approach to develop high quality Web Services.

Acknowledgments

We acknowledge the funding by ANPCyT (PICT-2012-0045) and CONICET (PIP 2013-2015, 11220120100185CO).

References

[1] M. Sellami, O. Bouchaala, W. Gaaloul, and S. Tata, "Communities of web service registries: Construction and management," *Journal of Systems and Software*, vol. 86, no. 3, pp. 835–853, March 2013. [Online]. Available: https://doi.org/10.1016/j.jss.2012.11.019

[2] C. Mateos, M. Crasso, A. Zunino, and J. L. Ordiales Coscia, "Revising WSDL Documents: Why and How - Part II," *IEEE Internet Computing*, vol. 17, no. 5, pp. 46–53, September/October 2013. [Online]. Available: http://dx.doi.org/10.1109/MIC.2013.4

[3] M. Crasso, A. Zunino, and M. Campo, "A survey of approaches to Web Service discovery in Service-Oriented Architectures," *Journal of Database Management*, vol. 22, no. 1, pp. 103–134, January 2011. [Online]. Available: https://doi.org/10.4018/jdm.2011010105

[4] H. M. Sneed, "Measuring Web Service interfaces," in *12th IEEE International Symposium on Web Systems Evolution (WSE)*, September 2010, pp. 111–115. [Online]. Available: https://doi.org/10.1109/WSE.2010.5623580

[5] J. M. Rodriguez, M. Crasso, and A. Zunino, "An approach for web service discoverability anti-patterns detection," *Journal of Web Engineering*, vol. 12, no. 1–2, pp. 131–158, February 2013. [Online]. Available: http://www.rintonpress.com/journals/jweonline.html#v12n12

[6] D. Baski and S. Misra, "Metrics suite for maintainability of extensible markup language Web Services," *IET Software*, vol. 5, no. 3, pp. 320–341, June 2011. [Online]. Available: http://dx.doi.org/10.1049/iet-sen.2010.0089

[7] J. L. Ordiales Coscia, C. Mateos, M. Crasso, and A. Zunino, “Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools,” *International Journal of Web and Grid Services*, vol. 9, no. 2, pp. 107–126, 2013. [Online]. Available: https://doi.org/10.1504/IJWGS.2013.054108

[8] The Apache Software Foundation, “Web Services - Axis,” 2005. [Online]. Available: http://ws.apache.org/axis/java/user-guide.html#Java2WSDLBuildingWSDLFromJava

[9] ——, “Java to WS,” 2014. [Online]. Available: http://cxf.apache.org/docs/java-to-ws.html

[10] OW2 Consortium, “Easy WSDL toolbox,” 2014. [Online]. Available: http://easywsdl.ow2.org

[11] JBoss.org, “JBossWS - wsprovide,” 2014. [Online]. Available: https://developer.jboss.org/wiki/JBossWS-wsprovide?_sscc=t

[12] D. E. Geetha, T. S. Kumar, and K. R. Kanth, “Predicting the software performance during feasibility study,” *IET Software*, vol. 5, no. 2, pp. 201–215, April 2011. [Online]. Available: http://dx.doi.org/10.1049/iet-sen.2010.0075

[13] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Berkeley, California, USA: Addison-Wesley Longman, Inc., 1999.

[14] J. L. Ordiales Coscia, M. Crasso, C. Mateos, A. Zunino, and S. Misra, “Predicting Web Service maintainability via Object-Oriented metrics: A statistics-based approach,” in *Computational Science and Its Applications (ICCSA 2012)*, ser. Lecture Notes in Computer Science. Brazil: Springer-Verlag Berlin, Heidelberg, June 2012, vol. 7336, pp. 29–39. [Online]. Available: https://doi.org/10.1007/978-3-642-31128-4_3

[15] C. Mateos, A. Zunino, S. Misra, D. Anabalon, and A. Flores, “Keeping web service interface complexity low using an oo metric-based early approach,” in *42th Latin American Computing Conference (CLEI)*. Valparaiso, Chile: IEEE Computer Society Press, October 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/CLEI.2016.7833366

[16] S. Chidamber and C. Kemerer, “A metrics suite for Object Oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994. [Online]. Available: https://doi.org/10.1109/32.295895

[17] D. Baski and S. Misra, “Measuring and evaluating a design complexity metric for XML schema documents,” *Journal of Information Science and Engineering*, vol. 25, pp. 1405–1425, March 2009. [Online]. Available: http://eprints.covenantuniversity.edu.ng/776/

[18] J. Bansiya and C. G. Davis, “A hierarchical model for Object-Oriented design quality assessment,” *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, August 2002. [Online]. Available: https://doi.org/10.1109/32.979986

[19] R. Subramanyam and M. S. Krishnan, “Empirical analysis of ck metrics for Object-Oriented design complexity: Implications for software defects,” *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, April 2003. [Online]. Available: https://doi.org/10.1109/TSE.2003.1191795

[20] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of Object-Oriented metrics on open source software for fault prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, November 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.112

[21] C. Catal, “Software fault prediction: A literature review and current trends,” *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626–4636, April 2011. [Online]. Available: https://doi.org/10.1016/j.eswa.2010.10.024

[22] P. Meirelles, C. Santos, J. Miranda, F. Kon, A. Terceiro, and C. Chavez, “A study of the relationships between source code metrics and attractiveness in free software projects,” in *Brazilian Symposium on Software Engineering (SBES’10)*. Brazil: IEEE Computer Society, October 2010, pp. 11–20. [Online]. Available: https://doi.org/10.1109/SBES.2010.27

[23] E. J. Weyuker, “Evaluating software complexity measures,” *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, September 1988. [Online]. Available: https://doi.org/10.1109/32.6178

[24] B. Kitchenham, S. L. Pfleeger, and N. Fenton, “Towards a framework for software measurement validation,” *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929–944, December 1995. [Online]. Available: https://doi.org/10.1109/32.489070

[25] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, December 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233837

[27] D. Spinellis, "Tool writing: A forgotten art?" *IEEE Software*, vol. 22, pp. 9–11, July 2005. [Online]. Available: https://doi.org/10.1109/MS.2005.111

[28] P. Arabie, L. Hubert, and G. De Soete, *Clustering and classification*. Dallas, TX, USA: World Scientific Publishing Co., 1996.

[29] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, October 1996. [Online]. Available: https://doi.org/10.1109/32.544352

[30] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1–7, September 2011. [Online]. Available: https://doi.org/10.1145/2020976.2020983

[31] M. Crasso, C. Mateos, A. Zunino, S. Misra, and P. Polvorín, "Assessing cognitive complexity in java-based object-oriented systems: Metrics and tool support," *Computing and Informatics*, vol. 35, no. 3, pp. 497–527, July 2016. [Online]. Available: http://www.cai.sk/ojs/index.php/cai/article/viewArticle/1747

[32] P. Sripairojthikoon and T. Senivongse, "Concept-based readability of web services descriptions," in *15th International Conference on Advanced Communication Technology (ICACT)*, PyeongChang, South Korea, March 2013, pp. 853–858. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6488317/

[33] K. Becker, J. Pruyne, S. Singhal, A. Lopes, and D. Milojicic, "Automatic determination of compatibility in evolving services," *International Journal of Web Services Research*, vol. 8, pp. 21–40, January 2011. [Online]. Available: https://doi.org/10.4018/jwsr.2011010102

[34] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinneide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. 10, no. 04, pp. 603–617, July-August 2017. [Online]. Available: https://doi.org/10.1109/TSC.2015.2502595

[35] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 23:1–23:53, August 2016. [Online]. Available: https://doi.org/10.1145/2932631

[36] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, December 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9414-4

[37] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–11:61, November 2012. [Online]. Available: https://doi.org/10.1145/2379776.2379787