SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 11, no. 1, pp. 31-48 (2012)

Avoiding WSDL Bad Practices in Code-First Web Services♦

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

[1] ISISTAN Research Institute.
[2]Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN)
Campus Universitario, Tandil (B7001BBO)
Buenos Aires, Argentina
[3]CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
e-mail: cmateos@conicet.gov.ar
Tel. +54-2293-440363, ext. 35

Abstract

Service-Oriented Computing allows software developers to structure applications as a set of standalone and reusable components called services. The common technological choice for materializing these services is Web Services, whose exposed functionality is described by using the Web Services Description Language (WSDL). Methodologically, Web Services are often built by first implementing their behavior and then generating the corresponding WSDL document via automatic tools. Good WSDL designs are crucial to derive reusable Web Services. We found that there is a high correlation between well-known Object-Oriented metrics taken in the code implementing services and the occurrences of bad design practices in their WSDL documents. This paper shows that some refactorings performed early when developing Web Services can greatly improve the quality of generated WSDL documents.

Keywords: Service-Oriented Computing; Web Services; Code-First; Object-Oriented Metrics; Wsdl Anti-Patterns; Early Detection.

1 Introduction

The success encountered by the Internet encourages practitioners, companies and governments to create software that consumes information and services that third-parties have publicly offered in the Web. Service-Oriented Computing (SOC) is a relatively new computing paradigm that supports the development of distributed applications in heterogeneous environments (Erickson & Siau, 2008) and has radically changed the way applications are architected, designed and implemented (Mateos, Crasso, Zunino, & Campo, 2010). The SOC paradigm introduces a new kind of building block called service, which represents functionality that is delivered by external providers (e.g.

♦ This article is an extended version of the paper presented in the 11th Argentine Symposium on Software Engineering (ASSE2011) - 39th JAIIO

a business or an organization), made available in registries, and remotely consumed using standard protocols. Far from being a buzzword, SOC has been exploited by major players in the software industry including Microsoft, Oracle, Google and Amazon.

The term "Web Services" refers to a stack of technologies for implementing the SOC paradigm. Web Services are services with well-defined interfaces that can be published, located and consumed by means of ubiquitous Web protocols (Erickson & Siau, 2008) such as SOAP (Consortium, 2007), i.e. this stack consists of Web-based technologies. Regarding Web Services interfaces, a provider describes them using the Web Services Description Language (WSDL), an XML-based language designed for specifying services' functionality as a set of abstract operations with inputs and outputs, and to associate binding information so that consumers can invoke the offered operations. The interactions between service producers, registry and consumers are shown in Figure 1.

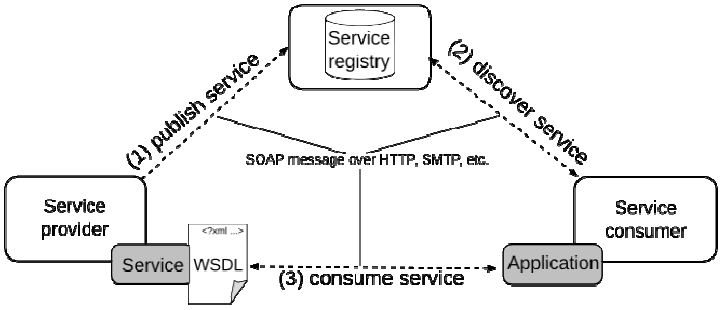

Figure 1 The Web Services model

Since back in mid-2002, to make their WSDL documents publicly available providers employed a specification of service registries called Universal Description, Discovery and Integration (UDDI), whose central purpose is to maintain meta-data about Web Services. Apart from this, UDDI defines an inquiry Application Programming Interface (API) for discovering services, which allows consumers to discover services that match their functional needs. Concretely, the inquiry API receives a keyword-based query and in turn returns a list of candidate WSDL documents, which the consumer who performs the discovery process must analyze. As a complement to UDDI, several syntactic Web Service registries such as Woogle (Dong, Halevy, Madhavan, Nemes, & Zhang, 2004), WSQBE (Crasso, Zunino, & Campo, 2008) and seekda![1] have emerged. These registries basically work by applying text processing or machine learning techniques, such as XML supervised classification (Crasso, Zunino, & Campo, 2008) or clustering (Rusu, Rahayu, & Taniar, 2008), to improve the retrieval effectiveness of the same keyword-based discovery process (Crasso, Zunino, & Campo, 2011).

Certainly, service interface design plays one of the most important roles in enabling third-party consumers to understand, discover and reuse services (Crasso, Rodriguez, Zunino, & Campo, 2010d). On one hand, unless appropriately specified by providers, service interface meta-data can be counterproductive and obscure the purpose of a service, thus hindering its adoption. Indeed, it has been shown that service consumers, when faced with two or more WSDL documents that are similar from a functional perspective, they tend to choose the most concisely described one (Crasso, Rodriguez, Zunino, & Campo, 2010a). Moreover, a WSDL description without many comments of its operations can make the associated Web Service difficult to be discovered (Crasso, Rodriguez, Zunino, & Campo, 2010a). Particularly, discovery precision of syntactic registries is harmed when dealing with poorly described WSDL documents (Crasso, Rodriguez, Zunino, & Campo, 2010a).

The work of (Crasso, Rodriguez, Zunino, & Campo, 2010a) integrally studies common discoverability bad practices, or *anti-patterns* for short, found in public WSDL documents, covering the problems mentioned in the previous paragraph. In (Crasso, Rodriguez, Zunino, & Campo, 2010c), the same authors provide a set of guidelines service providers should take into account when specifying service interfaces in order to obtain clear, discoverable

[1] Seekda!, http://webservices.seekda.com

services. However, a requirement inherent to applying these guidelines is that services are mostly built in a *contract-first* manner, a method that encourages designers to first derive the WSDL document of a service and then supply an implementation for it. Then, (Crasso, Rodriguez, Zunino, & Campo, 2010c) help providers in detecting and removing anti-patterns. However, the most used approach to build Web Services by the industry is *code-first*, which means that one first implements a service and then generates the corresponding WSDL document by automatically extracting and deriving the interface from the implemented code. Then, WSDL documents are not directly created by humans but are instead automatically derived via language-dependent tools. Consequently, anti-patterns may manifest themselves in the resulting WSDL documents when bad implementation practices are followed (Crasso, Rodriguez, Zunino, & Campo, 2010d) or deficient WSDL generation tools are used.

In this paper, we study the feasibility of avoiding these anti-patterns by using Object-Oriented (OO) metrics from the code implementing services. Basically, the idea is employing these metrics as "indicators" that warn the service developer about the potential occurrence of anti-patterns early during the Web Service implementation phase. In this way, this approach would benefit most software practitioners in the industry, which usually rely on code-first service construction. Specifically, through some statistical analysis, we found that a small sub-set of the OO metrics studied is highly correlated to the studied anti-patterns. Based on this, we analyze several simple code refactorings that developers can use to avoid anti-patterns in their service interfaces.

The rest of the paper is structured as follows. Section 2 gives some background on the WSDL anti-patterns. Then, Section 3 presents hypotheses for correlating these anti-patterns and metrics taken at the service implementation phase. Later, Section 4 presents experiments that evidence such correlations, the derived source code refactorings, and the positive effects of these latter in the WSDL documents. Section 5 surveys relevant related works. Section 6 concludes the paper.

2 Background

WSDL allows providers to describe two parts of a service, namely what it does (its functionality) and how to invoke it. The former part reveals the service interface that is offered to potential consumers. The latter part specifies technological aspects, such as transport protocols and network addresses. Consumers use the functional descriptions to match third-party services against their needs, and the technological details to invoke the selected service. With WSDL, service functionality is described as a *port-type* $W = \{O_0(I_0, R_0), .., O_N(I_N, R_N)\}$, which arranges different operations O_i that exchange input and return messages, I_i and R_i respectively. Main WSDL elements, such as *port-types*, *operations* and *messages*, must be labeled with unique names. Optionally, these WSDL elements might contain documentation in the form of comments.

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

Commonly, a WSDL document is the only publicly available meta-data that describes a Web Service. Thus, many approaches to Web Service discovery are based on service descriptions specified in WSDL (Crasso, Zunino, & Campo, 2011). Strongly inspired by classic Information Retrieval techniques, such as word sense disambiguation, stop-words removal, and stemming, these approaches extract keywords from WSDL documents, and then model extracted information on inverted indexes or vector spaces (Crasso, Zunino, & Campo, 2011). Then, generated models are employed for retrieving relevant service descriptions, i.e. WSDL documents, for a given keyword-based query. Different experiments empirically have confirmed that these approaches to discover services are very interesting, however as they rely on the descriptiveness of service specifications their retrieval effectiveness may be deteriorated by poorly written WSDL documents.

The work published in (Crasso, Rodriguez, Zunino, & Campo, 2010a) introduces the WSDL discoverability anti-patterns (see Table 1 for a brief description), measures their impact on both service retrieval effectiveness and human users' experience, and proposes refactoring actions to remedy the identified problems. The authors classify the identified bad practices as problems concerning how a service interface has been designed, problems on the comments and identifiers used to describe a service, and problems on how the data exchanged by a service are modeled. Each bad practice description is accompanied by a reproducible solution in (Crasso, Rodriguez, Zunino, & Campo, 2010c), thus they are called WSDL discoverability anti-patterns, or anti-patterns for short. A requirement inherent to apply these solutions is that services are built in a *contract-first* manner, a method that encourages designers to first derive the WSDL interface of a service and then supply an implementation for it using any programming language. Although with this method providers achieve the real importance of WSDL documents as a communication artifact, contract-first is not very popular among developers because the effort it requires is rather bigger than the required by its counterpart, namely code-first. Code-first means that one first implements a service and then generates the corresponding service contract by automatically extracting and deriving the interface from the implemented code. To understand how this works, let us take the case of Java2WSDL, a software tool that given a Java class produces a WSDL document with operations standing for all public methods declared in the class. Moreover, Java2WSDL associates an XML representation with each input/output method parameter -primitive types or objects- in XSD. One consequence of this WSDL generation method is that any change introduced in service implementations requires the re-generation of WSDL documents, which in turn may affect service consumers as service interfaces potentially change. In the end, developers focus on developing and maintaining service implementations, while delegating WSDL documents generation to code-first tools during service deployment.

Anti-pattern	**Occurs when**
Ambiguous names (AP_1)	Ambiguous or meaningless names are used for the main elements of a WSDL document.
Empty messages (AP_2)	Empty messages are used in operations that neither produce outputs nor receive inputs.
Enclosed data model (AP_3)	The data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD documents.
Low cohesive operations in the same port-type (AP_4)	Port-types have weak semantic cohesion.
Redundant data models (AP_5)	A WSDL document relies on many data-types for representing the same domain objects.
Whatever types (AP_6)	A special data-type is used for representing any object of the problem domain.

Table 1: The core sub-set of the Web Service discoverability anti-patterns

The main hypothesis of this paper is that it is possible to detect WSDL anti-patterns *early* in the implementation phase by basing on classic API metrics gathered from service implementation and an understanding about how WSDL generation tools work. As explained in (Crasso, Rodriguez, Zunino, & Campo, 2010d), the anti-patterns are strongly associated with API design qualitative attributes, in the sense that some anti-patterns spring when well-established API design golden rules are broken. For instance, the AP_4 anti-pattern is to place semantically unrelated *operations* in the same *port-type*, although modules with high cohesion tend to be preferable, which is a well-known lesson learned from structured design. The goal of this paper is to detect WSDL discoverability anti-patterns previous to generate WSDL documents, but by basing on service implementations since the code-first method is meant to be supported.

3 Hypothesis statements for early WSDL anti-patterns detection

The proposed approach aims at allowing providers to prevent their WSDL documents from incurring in the WSDL anti-patterns presented in (Crasso, Rodriguez, Zunino, & Campo, 2010a) when following the code-first method for

building services. To do this, the approach is supported by two facts. First, the approach assumes that a typical code-first tool performs a mapping *T*, formally $T : C \rightarrow W$.

Mapping T from $C = \{M(I_0, R_0),.., M_N(I_N, R_N)\}$ or the front–end class implementing a service to $W = \{O_0(I_0, R_0),.., O_N(I_N, R_N)\}$ or the WSDL document describing the service, generates a WSDL document containing a *port-type* for the service implementation class, having as many *operations O* as public methods *M* are defined in the class. Moreover, each *operation* of *W* will be associated with one input *message I* and another return *message R*, while each *message* conveys an XSD type that stands for the parameters of the corresponding class method. Code-first tools like Java2WSDL, WSDL.exe, and gSOAP (Van Engelen & Gallivan, 2002) are based on a mapping *T* for generating WSDL documents from Java, C#, and C++, respectively, though each tool implements *T* in a particular manner mostly because of the different characteristics of the involved programming languages.

Figures 2 and 3 show the generation of a WSDL document for two similar Web Services, using Java2WSDL and WSDL.exe, respectively. It can be noted that the generation process for both tools is the same, i.e. the mapping *T* maps public method on the service code to an *operation* containing two *messages* in the WSDL document and these, in turn, are associated with an XSD type containing the parameters of that operation. There are, however, some minor differences between the two generated WSDL documents. For example, Java2WSDL generates only one port-type with all the operations of the Web Service, whereas WSDL.exe generates three port-types (one for each transport protocol). As we mentioned before, these differences are a result of the implementation each tool uses when applying the mapping to the service code.

Figure 2 WSDL generation in Java

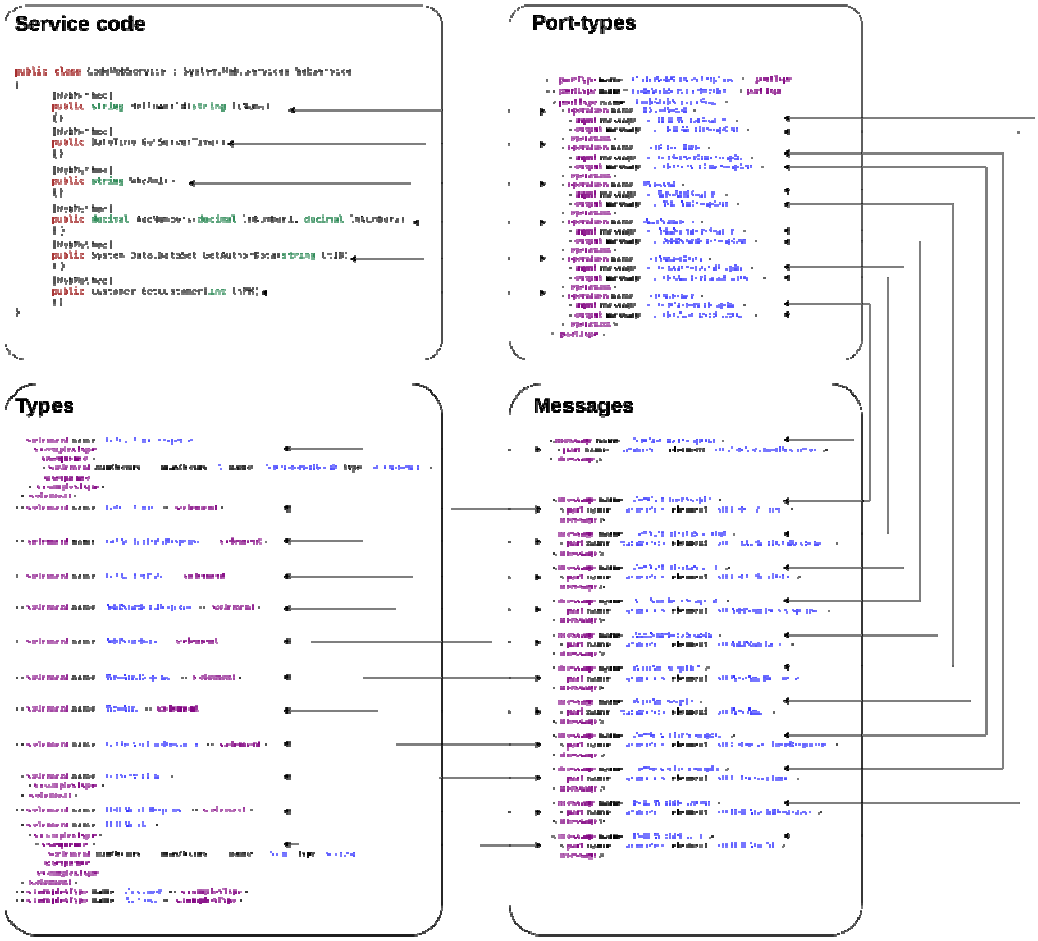

Figure 3 WSDL generation in C#

Furthermore, the second fact that underpins our approach is that WSDL discoverability anti-patterns are strongly associated with API design attributes (Crasso, Rodriguez, Zunino, & Campo, 2010d), which have been soundly studied by the software engineering community and as a result suites of related OO class-level metrics exist, such as the Chindamber and Kemerer's metric catalog (Chidamber & Kemerer, 1994). Consequently, these metrics tell providers about how a service implementation conforms to specific design attributes. For instance, the LCOM (Lack of Cohesion Methods) metric provides a mean to measure how well the methods of a class are semantically related to each other, while the *"Low cohesive operations in the same port-type"* anti-pattern measures WSDL *operations* cohesion. Here, the design attribute under study is cohesion, the metric is LCOM, and *"Low cohesive operations in the same port-type"* is the potentially associated anti-pattern.

By basing on the previous two facts, the idea behind the proposed approach is that by employing well-known software engineering metrics on a service code *C*, a provider might have an estimation of how the resulting WSDL document *W* will be like in terms of anti-pattern occurrences, since a known mapping *T* relates *C* with *W*. If indeed such metric/anti-pattern relationships exist, then it would be possible to determine a range of metric values for *C* so that *T* generates *W* without anti-patterns in the best case.

We established several hypotheses by using an exploratory approach to test the statistical correlation among OO metrics and the anti-patterns. For brevity and clarity, next we show the initial hypotheses that after the statistical analysis proved to hold.

Hypothesis 1 ($H_1 : CBO \rightarrow AP_3$). The higher the number of classes directly related to the class implementing a service (CBO metric), the more frequent the *Enclosed data model* anti-pattern occurrences.

Basically, CBO (Coupling Between Objects) (Chidamber & Kemerer, 1994) counts how many methods or instance variables defined by other classes are accessed by a given class. Code-first tools based on *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 data-types definitions within WSDL documents.

Hypothesis 2 ($H_2 : WMC \rightarrow AP_4$). The higher the number of public methods belonging to the class implementing a service (WMC metric), the more frequent the *Low cohesive operations in the same port-type* anti-pattern occurrences.

The WMC (Weighted Methods Per Class) (Chidamber & Kemerer, 1994) metric counts the methods of a class. We believe that a greater number of methods increases the probability that any pair of them are unrelated, i.e. having weak cohesion. Since *T*-based code-first tools map each method to an operation, a higher WMC may increase the possibility that resulting WSDL documents have low cohesive operations.

Hypothesis 3 ($H_3 : WMC \rightarrow AP_5$). The higher the number of public methods belonging to the class implementing a service (WMC metric), the more frequent the *Redundant data model*s anti-pattern occurrences.

The number of *message* elements defined within a WSDL document built under *T*-based code-first tools, is equal to the number of *operation* elements multiplied by two. As each *message* may be associated with a data-type, we believe that the likelihood of redundant data-type definitions increases with the number of public methods, since this in turn increase the number of *operation* elements.

Hypothesis 4 ($H_4 : WMC \rightarrow AP_1$). The higher the number of public methods belonging to the class implementing a service (WMC metric), the more frequent the *Ambiguous names* anti-pattern occurrences.

Similarly to H_3 , we believe that an increment in the number of methods may lift the number of non-representative names within a WSDL document, since for each method a *T*-based code-first tool automatically generates in principle five names (one for the operation, two for input/output messages, and two for data-types).

Hypothesis 5 ($H_5 : ATC \rightarrow AP_6$). The higher the number of method parameters belonging to the class implementing a service that are declared as non-concrete data-types (ATC metric), the more frequent the *Whatever types* anti-pattern occurrences.

ATC (Abstract Type Count) is a metric of our own that computes the number of method parameters that do not use concrete data-types, or use Java generics with type variables instantiated with non-concrete data-types. We have defined the ATC metric after noting that some *T*-based code-first tools map abstract data-types and badly defined generics to xsd:any constructors, which have been identified as root causes for the *Whatever types* anti-pattern (Pasley, 2006) (Crasso, Rodriguez, Zunino, & Campo, 2010a).

Hypothesis 6 ($H_6 : EPM \rightarrow AP_2$). The higher the number of public methods belonging to the class implementing a service that do not receive input parameters (EPM metric), the more frequent the *Empty messages* anti-pattern occurrences.

Similarly to ATC, we designed the EPM (Empty Parameters Methods) metric to count the number of methods in a class that do not receive parameters. We believe that increasing the number of methods without parameters may increase the likelihood of the *Empty messages* anti-pattern occurrences, because *T*-based code-first tools map this kind of methods onto an operation associated with one input *message* element not conveying XML data.

The next section describes the experiments that were carried out to test these six hypotheses as well as the relation between other OO metrics not included in the above list and the studied anti-patterns.

4 Statistical analysis and experiments

The approach chosen for testing the hypotheses of the previous section consists on gathering OO metrics from open source Web Services, and checking the values obtained against the number of anti-patterns found in services WSDL documents, using correlation methods to validate the usefulness of these metrics for anti-pattern prediction. To perform the analysis, we first implemented the software pipeline depicted in Figure 4. Basically, the input to this pipeline was a Web Service data-set that contained, for each service, its implementation code and dependency libraries needed for compiling and generating WSDL documents. The output, on the other hand, was a detailed per-service report of the statistical correlation between object-oriented metrics taken on the implementation code and anti-pattern occurrences calculated on the WSDL documents. It is worth noting that both the software and the data-set used in the experiments are available upon request.

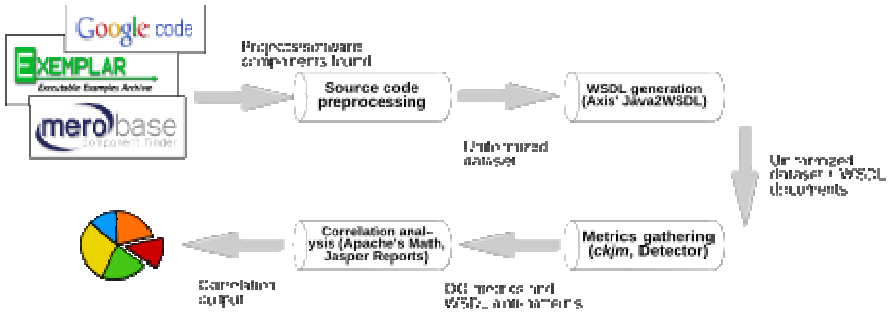

Figure 4 Software configuration used in the experiments

The described pipeline has been implemented using software tools for automatizing metrics recollection and anti-patterns detection, since metrics recollection is an extremely sensitive task for this experiment, but also a task that would require a huge amount of time to be manually carried on. Besides being a time consuming task, it was an error prone task. Therefore, we extended *ckjm* (Spinellis, 2005), a Java-based tool that computes a sub-set of the Chidamber-Kemerer metrics (Chidamber & Kemerer, 1994).

To measure the number of anti-patterns, we employed an automatic WSDL anti-pattern detection tool (Crasso, Rodriguez, Zunino, & Campo, 2010b). The WSDL Anti-patterns Detector (Crasso, Rodriguez, Zunino, & Campo, 2010b), or Detector for short, is a software whose purpose is automatically checking whether a WSDL document suffers from the anti-patterns of (Crasso, Rodriguez, Zunino, & Campo, 2010a) or not. The Detector receives a given WSDL document as input, and uses heuristics for returning a list of anti-pattern occurrences. As these heuristics are based on the different anti-pattern definitions, there are two groups of heuristics, namely Evident and Not immediately apparent. The Evident heuristics deal with those anti-patterns that can be detected by analyzing only the structure of WSDL documents, like *Empty Messages*, *Enclosed data-types*, *Redundant data models*, and *Whatever types* anti-patterns. The Not immediately apparent heuristics deal with detecting *Low cohesive operations in the same port-type* and *Ambiguous names* anti-patterns because they require a semantic analysis of the names and comments present in WSDL documents. As explained in (Crasso, Rodriguez, Zunino, & Campo, 2010b), the authors combine machine learning and natural processing language techniques to detect the anti-patterns of the second group.

In the tests, we used a data-set of 154 different real services whose implementations were collected via two code search engines, namely the Merobase component finder (http://merobase.com) and the Exemplar engine (Grechanik, Fu, Xie, McMillan, Poshyvanyk, & Cumby, 2010). Merobase allows users to harvest software components from a large variety of sources (e.g. Apache, SourceForge, and Java.net) and has the unique feature of supporting interface-driven searches, i.e. searches based on the abstract interface that a component should offer, apart from that of based on the text in its source code. On the other hand, Exemplar relies on a hybrid approach to keyword-based search that combines the benefits of textual processing and intrinsic qualities of code to mine repositories and consequently returns complete projects. Complementary, we collected projects from Google Code. All in all, the generated data-set provided the means to perform a significant evaluation in the sense that the different Web Service implementations came from real-life software engineers.
After collecting the components and projects, we uniformized the associated services by explicitly providing a Java interface in order to facade their implementations. Each WSDL document was obtained by feeding Axis' Java2WSDL[2] with the corresponding interface. Finally, the correlation analysis was performed by using Apache's Commons Math library[3], and plots were obtained via JasperReports[4].

The rest of the Section is structured as follows. Section 4.1 describes the statistical correlation results between OO metrics and anti-patterns that were obtained for the above data-set. Lastly, Section 4.2 explores several service refactorings at the source code level and their effect on the anti-patterns of resulting WSDL documents.

4.1 Object-Oriented metrics and WSDL anti-patterns: Correlation analysis

The commonest way of analyzing the empirical relation between independent and dependent variables is by defining and statistically testing experimental hypotheses (Fenton & Pfleeger, 1998). In this sense, we set the 6 anti-patterns described up to now as the dependent variables, whose values were produced by using the Detector, while we used OO metrics as the independent variables, which were computed via the *ckjm* tool.

We used 11 metrics for measuring services implementations, which played the role of independent variables. WMC, CBO, RFC, and LCOM have been selected from the work of Chindamber and Kemerer (Chidamber & Kemerer, 1994). The WMC (Weighted Methods Per Class) metric counts the methods of a class. CBO (Coupling Between Objects) counts how many methods or instance variables defined by other classes are accessed by a given class. RFC (Response for Class) counts the methods that can potentially be executed in response to a message received by an object of a given class. LCOM (Lack of Cohesion Methods) provides a mean to measure how well the methods of a class are related to each other, with higher values of the metric standing for less cohesive methods. From the work of Bansiya and Davis (Bansiya & Davis, 2002) we picked CAM (Cohesion Among Methods of Class) metric. CAM computes the relatedness among methods based upon the parameter list of these methods. Additionally, we used a number of ad-hoc measures we thought could be related to the WSDL metrics or studied anti-patterns, namely TPC (Total Parameter Count), APC (Average Parameter Count), ATC (Abstract Type Count), VTC (Void Type Count), and EPM (Empty Parameters Methods). The last employed metric was the well-known lines of code (LOC) metric.

The descriptive statistics for the anti-patterns and metrics studied are shown in Table 2. These values will be useful to help us interpret the results of the analysis throughout this section. In addition, they will facilitate comparisons against results from future similar studies.

We used the Spearman's rank correlation coefficient in order to establish the existing relations between the two kinds of variables of our model, i.e. the OO metrics (independent variables) and the anti-patterns (dependent variables). Table 3 depicts the correlation factors among the studied OO metrics. The cell values in bold are those coefficients which are statistically significant at the 5% level, i.e. p-value < 0.05, which is a common choice when performing statistical studies (Stigler, 2008). The sign of the correlation coefficients defines the direction of the

[2] http://ws.apache.org/axis/java

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

[4] JasperReports, http://jasperforge.org/projects/

relationship, i.e. positive or negative. A positive relation means that when the independent variable grows, the dependent variable grows too, and when the independent variable falls the dependent goes down as well. Instead, a negative relation means that when independent variables grow, the dependent metrics fall, and vice versa. The absolute value, or correlation factor, indicates the intensiveness of the relation regardless of its sign. The correlation factors depicted in Table 3 clearly show that the metrics studied are not statistically independent and, therefore, capture redundant information. In other words, if a group of variables in a data-set are strongly correlated, these variables are likely to measure the same underlying dimension (i.e. cohesion, complexity, coupling, etc.). In the case of our study, it can be seen from Table 3 that the metrics WMC, RFC, LOC and LCOM have a perfect correlation, i.e. |correlation factor| = 1, and therefore only one of them needs to be considered. Given that WMC is more popular among developers and is better supported in IDE tools compared to the other three, we chose to exclude the latter from further analysis and focus on WMC instead.

Metrics	**Minimum**	**Maximum**	**Mean**	**Std. Dev**
WMC	1.00	97.00	5.73	11.13
CBO	0.00	27.00	2.02	2.91
RFC	1.00	97.00	5.73	11.13
LCOM	0.00	4656.00	75.21	427.42
LOC	1.00	97.00	5.73	11.13
CAM	0.13	1.00	0.78	0.23
TPC	0.00	228.00	10.91	24.23
APC	0.00	17.00	2.04	1.83
ATC	0.00	20.00	1.09	2.25
VTC	0.00	25.00	1.05	3.53
EPM	0.00	11.00	0.57	1.64
Ambiguous names (AP_1)	1.00	243.00	13.43	28.77
Empty messages (AP_2)	0.00	11.00	0.57	1.64
Enclosed data model (AP_3)	0.00	44.00	5.41	7.09
Low cohesive operations in the same port-type (AP_4)	0.00	910.00	9.78	74.99
Redundant data models (AP_5)	0.00	891.00	23.46	100.46
Whatever types (AP_6)	0.00	17.00	0.92	1.86

Table 2: Descriptive statistics

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

Table 3: Correlation among OO metrics

Table 4 shows the correlation between the considered OO metrics and the anti-patterns, namely *Ambiguous names* (AP_1), *Empty messages* (AP_2), *Enclosed data model* (AP_3), *Low cohesive operations in the same port-type* (AP_4), *Redundant data models* (AP_5) and *Whatever types* (AP_6). The factors in bold represent those with a p-value < 0.05. From the table, it can be observed that there is a high statistical correlation between a sub-set of the analyzed metrics and the anti-patterns. Concretely, two out of the eleven metrics, i.e. WMC and CBO, are positively correlated to four of the six studied anti-patterns, i.e. *Ambiguous names* (AP_1), *Enclosed data model* (AP_3), *Low cohesive operations in the same port-type* (AP_4) and *Redundant data models* (AP_5). Additionally, ATC and EPM are the best predictors for the two remaining anti-patterns, i.e. *Empty messages* (AP_2) and *Whatever types* (AP_6).

Anti-patterns / Metrics	**WMC**	**CBO**	**CAM**	**TPC**	**APC**	**ATC**	**VTC**	**EPM**
AP_1	**0.91**	**0.29**	**-0.75**	**0.76**	0.02	**0.20**	0.09	**0.21**
AP_2	**0.41**	-0.15	**-0.36**	0.08	**-0.33**	**-0.18**	**0.37**	**1.0**
AP_3	0.08	**0.92**	**-0.22**	**0.19**	**0.27**	**0.48**	-0.12	**-0.21**
AP_4	**0.64**	0.11	**-0.56**	**0.51**	-0.004	0.11	**0.40**	**0.40**
AP_5	**0.87**	0.13	**-0.63**	**0.65**	-0.11	0.06	0.08	**0.25**
AP_6	**0.17**	**0.43**	**-0.23**	**0.33**	**0.31**	**0.96**	-0.02	**-0.24**

Table 4: Correlation between OO metrics and anti-patterns

By looking at Table 4 one could state that there is a high statistical correlation between eight of the analyzed OO metrics and, at least, one anti-pattern. Initially, this implies that these independent variables could be somehow "controlled" by software engineers attempting to obtain better WSDL documents, in terms of anti-patterns. However, as determining the best set of *controllable* independent variables would deserve a deeper analysis, we will focus on determining a minimalist sub-set of OO metrics for this paper. This does not only improve the readability of the results but, as will be explained in Section 4.2, in order to avoid WSDL anti-patterns, early code refactorings by basing on OO metrics values are necessary. Thus, the smaller the number of considered OO metrics upon refactoring, the more simple (but still effective) this refactoring process becomes. Therefore, the 8x6 Table 4 may be reduced into a new 4x6 table (see Table 5), which represents a minimalist sub-set of correlated metrics.

Anti-patterns / Metrics	**WMC**	**CBO**	**ATC**	**EPM**
AP_1	**0.91**	-	-	-
AP_2	**-**	-	-	**1.00**
AP_3	-	**0.92**	-	-
AP_4	**0.64**	-	-	-
AP_5	**0.88**	-	-	-
AP_6	-	-	**0.97**	**-**

Table 5: Strongest correlations between OO metrics and anti-patterns

To do this, it is worth noting that there are two other OO metrics that are highly correlated to several anti-patterns, namely CAM and TPC. However, as we showed in Table 3, these two metrics present high correlation factors with

WMC and therefore are likely to measure redundant information. Then, only WMC needs to be considered. Furthermore, if we want to keep only the highest correlated pairs of metrics and anti-patterns, the correlation factors below $|0.6|$ at the 5% level can be discarded. Then, the VTC and APC metrics can be excluded.

Additionally, for the sake of readability, we have employed a different approach to depict the correlation matrix of Table 5, which is shown in Figure 5. In the Figure, blank cells stand for not statistically significant correlations, whereas cells with circles represent correlation factors at the 5% level. The diameter of a circle represents a correlation factor, i.e. the bigger the correlation factor the bigger the diameter. The color of a circle stands for the correlation sign, being black used for positive correlations and white for negative ones. Furthermore, those cells representing each of the correlations proposed on the hypotheses defined in Section 3 show their associated names (H_1 through H_6). Then, it can be seen that the six bigger circles in the Figure, i.e. the six highest statistically significant correlation factors, correspond precisely with the six defined hypotheses, showing that they are supported by our data, thus confirming their validity.

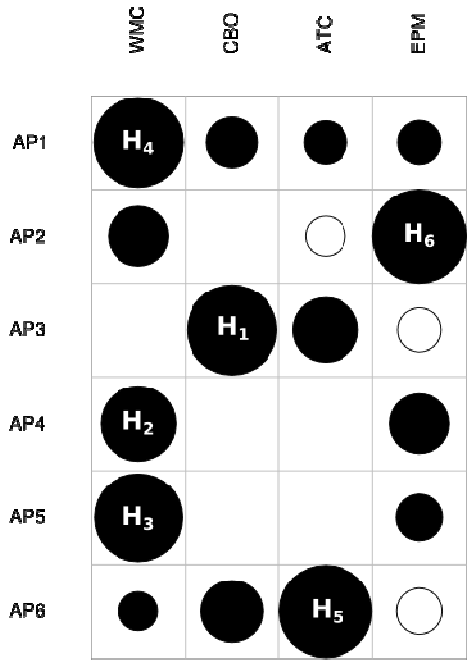

Figure 5: Graphical representation of the correlation between OO metrics and anti-patterns

4.2 Early code refactorings for improving WSDL documents

The correspondences between the minimalist set of OO metrics and the 6 anti-patterns, which were found to be statistically significant for the analyzed Web Service data-set suggest that, in practice, an increment/decrement of metric values taken on the code of a Web Service directly affects anti-pattern occurrences in its code-first generated WSDL. Then, we performed some source code refactorings driven by these metrics on our data-set so as to quantify the effect on anti-pattern occurrence. We conducted five rounds of refactoring which in turn produced five new data-sets, one for each of the four mentioned metrics and a fifth one where all the previous refactorings were included. Moreover, each refactoring was applied on the original data-set, meaning they are completely independent from each other. For the sake of brevity, in the rest of this section we will refer to these new refactored data-sets as RefactoredWMC (DS_1), RefactoredCBO (DS_2), RefactoredATC (DS_3), RefactoredEPM (DS_4) and RefactoredAll (DS_5). It is worth noting that each of these refactorings provides a practical way to test the validity of the hypotheses defined in Section 3, in the sense that a variation on each OO metric value should produce the same effect on its associated anti-patterns. For example, the RefactoredCBO (DS_2) data-set, which produces a

decrement of the CBO metric, is expected to have a lower number of occurrences of the *Enclosed data model* (AP_3) anti-pattern, since hypothesis H_1 stated that as the value of the OO metric increases so does the number of occurrences of the anti-pattern, thus suggesting a positive correlation between the two.

The first metric to consider was WMC. In this case we refactored the original data-set by splitting the services that contained more than one operation into two new services so that on average the metric in the refactored services represented a 50% of the original value. This refactoring resulted in a new data-set that contained approximately twice as many services as the original one.
Next, we focused on CBO by modifying the original services' implementation code to replace every occurrence of a complex data-type with the Java primitive type String. This refactoring did not modify the size of the resulting data-set with respect to the original one.

In a third refactoring round, we focused on the ATC metric, which computes the number of parameters in a class that are declared as data structures –i.e. collections– that do not use Java generics or as Java's root class **Object**. In the former case, when this practice is followed, these collections cannot be automatically mapped onto concrete XSD data-types for both the container collective data-type and the contained data-type in the final WSDL. A similar problem arises with parameters whose data-type is **Object**. In this sense, we modified the original services in order to reduce ATC by, basically, replacing generic arguments with concrete ones, which resulted in a data-set of equal size.

The last metric taken into consideration was EPM, which counts the number of methods in a class that do not receive input parameters. The refactoring applied in this case was to introduce a new boolean parameter to each of these methods. Finally, a last round of refactoring was performed by deriving a new data-set that included all the above mentioned code modifications. Tables 6 and 7 show the impact of the refactoring process on the OO metrics and the anti-patterns, respectively. For those metric on which the refactoring is focused, the cell value is in bold.

Metric	**Original (average)**	DS_1 **(average)**	DS_2 **(average)**	DS_3 **(average)**	DS_4 **(average)**	DS_5 **(average)**
WMC	8.64	**4.45**	8.64	8.64	8.64	**4.45**
CBO	2.02	1.39	**0.00**	2.02	2.02	**0.00**
ATC	1.11	0.59	0.68	**0.04**	1.11	**0.00**
EPM	0.94	0.44	0.94	0.94	**0.00**	**0.00**

Table 6: Refactoring: impact on OO metrics

Anti-pattern	**Original (avg.)**	DS_1 **(avg.)**	DS_2 **(avg.)**	DS_3 **(avg.)**	DS_4 **(avg.)**	DS_5 **(avg.)**
Ambiguous names (AP_1)	20.02	10.79	20.02	20.02	20.96	11.24
Empty messages (AP_2)	0.94	0.44	0.94	0.94	0.00	0.00
Enclosed data model (AP_3)	3.28	2.61	0.04	3.25	3.28	0.01
Low cohesive operations in the same port-type (AP_4)	24.62	8.19	24.62	24.62	19.04	6.25
Redundant data models (AP_5)	52.96	15.10	132.96	53.89	57.81	34.10
Whatever types (AP_6)	0.83	0.43	0.62	0.00	0.83	0.00
Total number of anti-patterns	102.66	37.58	179.21	102.72	101.92	51.59

Table 7: Refactoring: impact on anti-patterns

From the results presented it can be observed that a decrement on the values of the OO metrics produced the same effect on their associated anti-patterns. Concretely, reducing the value of WMC by 50% caused an average decrease of the *Ambiguous names* (AP_1), *Low cohesive operations in the same port-type* (AP_4) and *Redundant data models* (AP_5) anti-patterns occurrences of 46.09%, 66.74% and 71.48%, respectively. Similar results were obtained when refactoring the CBO, EPM and ATC metrics producing an average decrement of the *Enclosed data model* (AP_3), *Empty messages* (AP_2) and *Whatever types* (AP_6) anti-patterns by 98.85%, 100.00% and 100.00%, respectively. This provides practical evidence to better support part of the correlation analysis of the previous section.

It can also be observed that, while the individual metric refactorings had a positive impact on their associated anti-patterns, some of them also introduced increments in the number of occurrences of other anti-patterns. To clarify this, let us take for example the case of the CBO metric refactoring, which not only produced a decrement on the *Enclosed data model* (AP_3) anti-pattern, but also a considerable increment on the *Redundant data models* (AP_5) anti-pattern. Furthermore, the negative impact of this refactoring outweighs its benefits since the total number of anti-patterns is higher than those found in the original data-set. This kind of situations are known as *trade-offs*. As in software literature in general, here a trade-off represents a situation in which the software engineer should analyze and select among different metric-driven implementation alternatives. Two other metrics represent trade-off opportunities. By decreasing the ATC metric, resulting WSDL documents will present a smaller value for the *Whatever types* (AP_6) anti-pattern than the original WSDL document. However, this will cause an increment of the *Redundant data models* (AP_5) anti-pattern. A similar situation occurs with the EPM metric and the *Empty messages* (AP_2) and *Redundant data models* (AP_5) anti-patterns.

Interestingly, controlling the WMC metric is safe, in the sense that it does not present trade-off situations and by modifying its value no undesired collateral effects will be generated. Moreover, as shown in Table 6, when the WMC metric is refactored to reduce its value the rest of the OO metrics are indirectly affected and their values decrease as well. A consequence of this fact is that all the anti-patterns reduce their total number of occurrences, and not just those associated with WMC.

Finally, it is worth noting that when all the refactorings were applied on the same data-set (RefactoredAll) the total number of anti-patterns was reduced with respect to the original data-set but it was slightly higher than the one obtained by applying only the WMC refactoring. Considering that code refactoring is a time consuming process (Fowler, 1999), we can conclude that if the goal is to minimize the total number of anti-patterns then focusing only on WMC for the refactorings results in the most efficient choice since, as we mentioned previously, decreasing its value indirectly affects all the other OO metrics.
The results of the refactoring process presented in this section, particularly the fact that WMC is the metric that produces the lowest total number of anti-patterns when refactored, are consistent with the hypotheses defined in Section 3. Concretely, three of the six defined hypotheses, i.e. H_2 through H_4 , proposed a positive correlation between WMC and three anti-patterns, namely *Ambiguous names* (AP_1), *Low cohesive operations in the same port-type* (AP_4) and *Redundant data models* (AP_5). Furthermore, these anti-patterns present the highest average number of occurrences out of the six studied anti-patterns, as shown in Table 2. In contrast, the remaining three metrics, i.e. CBO, ATC and EPM, are associated with a single anti-pattern each, as stated in hypotheses H_1 , H_5 and H_6 , and they occur considerably less frequently on the original data-set. Then, their individual impact on the total number of anti-patterns is far less significant than the one from WMC. These results can be better observed in Figure 6.

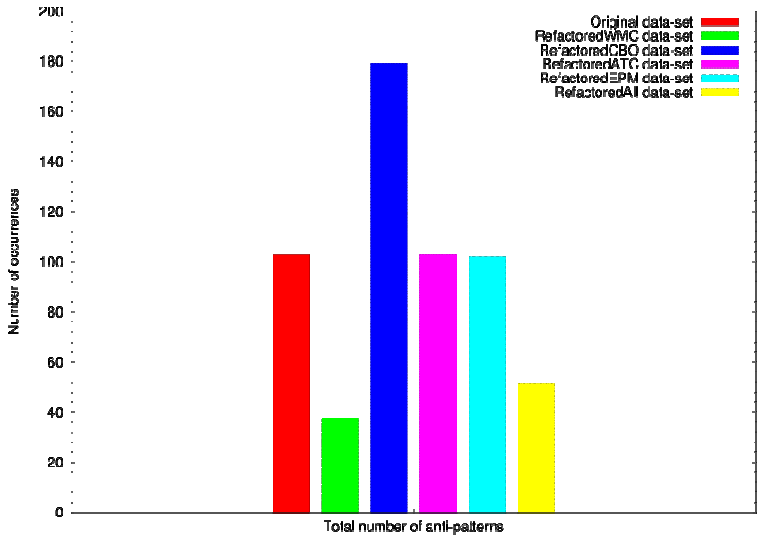

Figure 6 Refactoring: impact on the total number of anti-pattern occurrences

5 Related work

Certainly, our work is to some point related to a number of efforts that can be grouped into two broad classes. On the one hand, there is a substantial amount of research concerning improving services with respect to the quality of the contracts exposed to consumers (Fan & Kambhampati, 2005) (Blake & Nowlan, 2008) (Pasley, 2006) (Crasso, Rodriguez, Zunino, & Campo, 2010d) (Crasso, Rodriguez, Zunino, & Campo, 2010a). In particular, (Crasso, Rodriguez, Zunino, & Campo, 2010a) subsumes the research mentioned previously, and also supplies each identified problem with a practical solution, thus conforming a unified catalog of WSDL discoverability anti-patterns. The importance of these anti-patterns was measured by manually removing anti-patterns from a data-set of ca. 400 WSDL documents and comparing the retrieval effectiveness of several syntactic discovery mechanisms when using the original WSDL documents and the improved ones, i.e. the WSDL documents that have been refactored according to each anti-pattern solution. The fact that the results related to the improved data-sets surpass those achieved by using the original data-set regardless the approaches to service discovery employed, provides empirical evidence that suggests that the improvements are explained by the removal of discoverability anti-patterns rather than the incidence of the underlying discovery mechanism. Furthermore, the importance of WSDL discoverability anti-patterns has been increasingly emphasized in (Crasso, Rodriguez, Zunino, & Campo, 2010d), when the authors associate anti-patterns with software API design principles. In this sense, we can say that our approach is related to such efforts since we share the same goal, i.e. obtaining more legible, discoverable and clear service contracts.

On the other hand, in our approach, these aspects are quantified in the obtained contracts by means of specific WSDL-level metrics. Furthermore, we found that the values of such metrics can be "controlled" based on the values of OO metrics taken on the code implementing services prior to WSDL generation. Then, our approach is also related to some efforts that attempt to predict the value of quality metrics (e.g. number of bugs or popularity) in conventional software based on traditional OO metrics at implementation time (Subramanyam & Krishnan, 2003) (Gyimothy, Ferenc, & Siket, 2005) (Meirelles, Jr., Miranda, Kon, Terceiro, & Chavez, 2010). Particularly, our work is based on the results presented in (Ordiales, Mateos, Crasso, & Zunino, 2011) on which a public data-set of around 90 web services is statistically analyzed to determine whether or not a high correlation factor could be established between WSDL anti-patterns and OO metrics. The mentioned approach showed that there is significant statistical correlation between the two variables, and that by decreasing the values of some OO metrics certain anti-patterns are also reduced. However, one of the limitations of (Ordiales, Mateos, Crasso, & Zunino, 2011) is that only two metrics were taken into consideration for the refactoring rounds, i.e. WMC and ATC. Furthermore, the impact of these code modifications were not analyzed separately for each metric but combined on the same data-set. This, in turn, did not allow for a trade-off analysis between the different refactoring strategies.

6 Conclusions

Service contract design and particularly WSDL document specification, plays the most important role in enabling third-party consumers to understand, discover and reuse Web Services (Crasso, Rodriguez, Zunino, & Campo, 2010d). In previous research, it has been shown that Web Services have fewer chances of being reused unless some common WSDL discoverability anti-patterns are removed (Crasso, Rodriguez, Zunino, & Campo, 2010a). However, an inherent prerequisite for removing such anti-patterns is that services are built in a contract-fist manner, by which developers have more control on the WSDL of their services. Mostly, the industry is based on code-first Web Service development, which means that developers first derive a service implementation and then generate the corresponding service contracts from the implemented code.

In this paper, we have focused on the problem of how to obtain WSDL documents that are free from those undesirable anti-patterns when using code-first, independently of the generation tool used. Based on the approach followed by several existing works in which some quality attributes of the resulting software are predicted during development time and in particular on the approach presented in (Ordiales, Mateos, Crasso, & Zunino, 2011), we worked on the hypothesis that anti-pattern occurrences at the WSDL level can be avoided by basing on the value of OO metrics taken at the code implementing services. We used well-established statistical methods for coming out with the set of OO metrics that best correlate and explain anti-pattern occurrence by using a data-set of real Web Services. To validate these findings from a practical perspective, we also studied the effect of applying metric-driven code refactorings to some of the Web Services of the data-set on the anti-patterns in the generated WSDL documents. Interestingly, we found that these code refactorings effectively reduce anti-patterns, thus improving the resulting service contracts.

The evaluation of this work can be criticized at first sight, by basing on the fact that we employed only one code-first tool for the test. However, it is worth remarking that many code-first tools base on the same mapping function. Therefore, though the results cannot be generalized to all available code-first tools, the studied dependent variables are more likely to be affected by applying refactorings to service implementations rather than by changing the WSDL generation tool.

We are extending this work in several directions. On one hand, we are studying more refactorings, which in turn could be automated with the help of an IDE. As a starting point, we will use IntelliJ Idea (http://www.jetbrains.com/idea), a Java-based IDE that has many built-in refactoring functions and is designed to be extensible. Second, we will incorporate into our analysis less popular, but nevertheless other WSDL generation tools such as EasyWSDL and JBoss' *wsprovide*. The goal of this task is bringing our findings to a broader audience. Third, we will study the relationships between the anti-patterns and other OO metrics, including traditional metrics such as the ones proposed by Halstead, McCabe, or Henry and Kafura (Tsui & Karam, 2006), and at the same time newer ones (Al Dallal, 2011). This could in turn eventually lead to investigate the effect of other kind of refactorings.

Finally, another research line we are planning to work on relates to service discovery. It is known that, when developing contract-first Web Services, removing WSDL anti-patterns or at least reducing the number of their occurrences increases the retrieval efficiency of syntactic Web Service search engines and thus simplifies discovery (Crasso, Rodriguez, Zunino, & Campo, 2010c). In this context, anti-pattern avoidance is manually carried out by developers as they design the contract of their Web Services. In the approach presented in this paper, on the contrary, anti-patterns are removed or mitigated automatically and indirectly based on source code refactorings. In this sense, we will investigate the impact of the different refactorings and the extent to which they are applied in the effectiveness of service retrieval. Moreover, we have found and reported in this paper that there are trade-off situations where a refactoring for a particular OO metric may yield good results as measured by the number of some anti-pattern occurrences but bad results with respect to other anti-patterns. Then, a specific refactoring might have a positive impact on the total number of anti-patterns but produce the opposite effect on service discovery. Therefore, addressing this issue is subject of further research.

Acknowledgments

We acknowledge the financial support provided by ANPCyT (PAE-PICT 2007-02311).

References

Al Dallal, J. (2011). Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics. *Software Engineering, IEEE Transactions on , 37* (6), 788-804.

Bansiya, J., & Davis, C. (2002). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering , 28*, 4-17.

Blake, M. B., & Nowlan, M. (2008). Taming Web Services from the Wild. *IEEE Internet Computing , 12*, 62-69.

Chidamber, S., & Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering , 20* (6), 476-493.

Consortium, W. (June de 2007). SOAP Version 1.2 Part 1: Messaging Framework. *SOAP Version 1.2 Part 1: Messaging Framework* .

Crasso, M., Rodriguez, J. M., Zunino, A., & Campo, M. (2010a). An analysis of frequent ways of making undiscoverable Web Service descriptions. *Electronic Journal of SADIO - Special issue of Software Enginneering in Argentina: Present and Future Trends (Extended version of selected papers ASSE 2009) , 9* (1), 5-23.

Crasso, M., Rodriguez, J. M., Zunino, A., & Campo, M. (2010b). Automatically Detecting Opportunities for Web Service Descriptions Improvement. En W. Cellary, & E. Estevez (Ed.). (págs. 139-150). Springer.

Crasso, M., Rodriguez, J. M., Zunino, A., & Campo, M. (2010c). Improving Web Service Descriptions for effective service discovery. *Science of Computer Programming , 75* (11), 1001-1021.

Crasso, M., Rodriguez, J. M., Zunino, A., & Campo, M. (2010d). Revising WSDL Documents: Why and How. *IEEE Internet Computing , 14* (5), 30-38.

Crasso, M., Zunino, A., & Campo, M. (2011). A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management , 22* (1), 103-134.

Crasso, M., Zunino, A., & Campo, M. (2008). Easy Web Service Discovery: A Query-By-Example Approach. *Science of Computer Programming , 71* (2), 144-164.

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., & Zhang, J. (2004). Similarity Search for Web Services. En M. A. Nascimento, M. T. {\"O}zsu, D. Kossmann, R. J. Miller, J. A. Blakeley, & K. B. Schiefer (Ed.). (págs. 372-383). Morgan Kaufmann.

Erickson, J., & Siau, K. (2008). Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating Hype from Reality. *Journal of Database Management , 19* (3), 42-54.

Fan, J., & Kambhampati, S. (2005). A snapshot of public Web Services. *SIGMOD Record , 34* (1), 24-32.

Fenton, N., & Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach* (2nd ed.). PWS Publishing Co.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., & Cumby, C. (2010). A search engine for finding highly relevant applications. (págs. 475-484). ACM Press.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering , 31* (10), 897-910.

Mateos, C., Crasso, M., Zunino, A., & Campo, M. (2010). Separation of Concerns in Service-Oriented Applications Based on Pervasive Design Patterns. *ACM Special Interest Group on Applied Computing* (págs. 849-853). ACM.

Meirelles, P., Jr., C. S., Miranda, J., Kon, F., Terceiro, A., & Chavez, C. (2010). A Study of the Relationships between Source Code Metrics and Attractiveness in Free Software Projects. *0*, págs. 11-20. IEEE Computer Society.

Ordiales, J. L., Mateos, C., Crasso, M., & Zunino, A. (2011). Avoiding WSDL Bad Practices in Code-First Web Services., (págs. 1-12).

Pasley, J. (2006). Avoid XML Schema Wildcards For Web Service Interfaces. *IEEE Internet Computing , 10*, 72-79.

Rusu, L., Rahayu, W., & Taniar, D. (2008). Intelligent Dynamic XML Documents Clustering. (págs. 449-456). IEEE Computer Society.

Spinellis, D. (2005). Tool Writing: A Forgotten Art? *IEEE Software , 22*, 9-11.

Stigler, S. (2008). Fisher and the 5% level. *Chance , 21*, 12-12.

Subramanyam, R., & Krishnan, M. (2003). Empirical Analysis of CK Metrics for Object-Oriented Design

Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering , 29* (4), 297-310.
Tsui, F. F., & Karam, O. (2006). *Essentials of Software Engineering*. Prentice Hall.
Van Engelen, R., & Gallivan, K. (2002). The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. (págs. 128-135). IEEE Computer Society.