Avoiding WSDL Bad Practices in Code-First Web Services

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

[1] ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682.
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

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 the WSDL anti-patterns 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

Service-Oriented Computing (SOC) is a relatively new computing paradigm that has radically changed the way applications are architected, designed and implemented [1]. The SOC paradigm introduces a new kind of building block called *service*, which represents functionality that is delivered by external providers (e.g. 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 the de facto standard for implementing the SOC paradigm. Web Services are *services* enabled by using ubiquitous Web protocols [2]. When using Web Services, a provider describes each service technical contract in 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.

To make their WSDL documents publicly available, providers usually employ 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 [3], WSQBE [4] and seekda![3] have emerged. These registries basically work by applying text processing or machine learning techniques, such as XML supervised classification [4] or clustering [5], to improve the retrieval effectiveness of the same keyword-based discovery process [6].

Certainly, service contract design plays one of the most important roles in enabling third-party consumers to understand, discover and reuse services [7]. On one hand, unless appropriately specified by providers, service contract 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 contracts in WSDL that are similar from a functional perspective, they tend to choose the most concisely described [8]. Moreover, a WSDL description without much comments of its operations can make the associated Web Service difficult to be discovered [8]. Particularly, discovery precision of syntactic registries is harmed when dealing with poorly described WSDL documents [8].

The work of [8] 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 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 user about the potential occurrence of anti-patterns early in the Web Service implementation phase. Specifically, through some statistical analysis, we found that there is a statistical significant, high correlation between several traditional and ad-hoc OO metrics and the anti-patterns. Based on this, we analyze several code refactorings that developers can use to avoid anti-patterns in their service contracts.

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

2 Background

WSDL is a language that allows providers to describe two parts of a service, namely what it does (its functionality) and how to invoke it. Following the version 1.1 of the WSDL specification, the former part reveals the service interface that is offered to potential consumers. The latter part specifies technological aspects, such as transport protocols and network addresses. Consumers use the functional descriptions to

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

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 define the structure of an XML element. 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 [6]. Strongly inspired by classic Information Retrieval techniques, such as word sense disambiguation, stop-words removal, and stemming, in general these approaches extract keywords from WSDL documents, and then model extracted information on inverted indexes or vector spaces [6]. 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, poorly written WSDL documents may deteriorate approaches retrieval effectiveness.

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

Anti-pattern	Occurs when
Ambiguous names	Ambiguous or meaningless names are used for the main elements of a WSDL document.
Empty messages	Empty messages are used in operations that do not produce outputs nor receive inputs.
Enclosed data model	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	Port-types have weak semantic cohesion.
Redundant data models	Many data-types for representing the same objects of the problem domain.
Whatever types	A special data-type is used for representing any object of the problem domain.

The work published in [8] 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. A requirement inherent to apply these actions is that services are

built in a *contract-first* manner, a method that encourages designers to first derive the WSDL contract of a service and then supply an implementation for it. However, the most used approach to build Web Services by the software industry is *code-first*, which 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 the best of our knowledge the relationship between the code-first approach and WSDL anti-patterns has not been studied until now.

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 [7], 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, one 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 [8] 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 frontend 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 WSDL.exe, Java2WSDL, and gSOAP [9] are based on a mapping T for generating WSDL documents from C#, Java and C++, respectively, though each tool implements T in a particular manner mostly because of the different characteristics of the involved programming languages.

Furthermore, the second fact underpinning our approach is that WSDL discoverability anti-patterns are strongly associated with API design attributes [7], which have been soundly studied by the software engineering community and as a result suites of related Object-Oriented (OO) class-level metrics exist, such as the Chindamber and Kemerer's metric catalog [10]. 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" 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). 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) [10] 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). 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) [10] 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). 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). 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). 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 [11,8].

Hypothesis 6 (H_6). 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 Statical 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 regression and correlation methods to validate the usefulness of these metrics for anti-pattern prediction. To perform the analysis, we first gathered a data-set that contained, for each service, its implementation code and dependency libraries needed for compiling and generating WSDL documents. A detailed per-service report of the statistical correlation between OO metrics taken on the implementation code and anti-pattern occurrences present in the WSDL documents was built. It is worth noting that both the software and the data-set used in the experiments are available upon request.

Report calculation has been automatized by using software tools for metrics recollection and anti-patterns detection, since the time needed to manually analyze a Web Service project was 2 days/developer and it is an error prone task. In the former case, we extended *ckjm* [12], a Java-based tool that computes a sub-set of the Chidamber-Kemerer metrics [10].

To measure the number of anti-patterns, we employed an automatic WSDL anti-pattern detection tool [13]. The WSDL Anti-patterns Detector [13], or Detector for short, is a software whose purpose is automatically checking whether a WSDL document suffers from the anti-patterns of [8] or not. The Detector receives a given WSDL document as input, and uses heuristics for returning a list of anti-pattern occurrences.

In the tests, we used a data-set of around 90 different real services whose implementation was collected via two code search engines, namely the Merobase component finder (`http://merobase.com`) and the Exemplar engine [14]. We also 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 developers.

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 with the corresponding interface. Finally, the correlation analysis was performed by using Apache's Commons Math library[4], and plots were obtained via JasperReports[5].

The rest of the Section is structured as follows. Section 4.1 describes the statistical correlation analysis between OO metrics and anti-patterns that were performed on 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 most common way of analyzing the empirical relation between independent and dependent variables is by defining and statistically testing experimental hypotheses [15]. 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.

Furthermore, we employed extra metrics, namely the LOC (Lines Of Code) metric, which counts the number of source code lines in a class (including comments), and two metrics from the work by Bansiya and Davis [16], i.e. DAM (Data Access Metric) and CAM (Cohesion Among Methods of Class). DAM gives a hint on data encapsulation by computing the ratio of the number of private (protected) attributes to the total number of attributes declared in a class, while CAM computes the relatedness among methods based upon the parameter list of these methods. We also included in our study the Morris' AMC (Average Method Complexity) metric [17], i.e. the sum of the cyclomatic complexity of all methods divided by the total number of methods in a class. Finally, as suggested earlier, we extended *ckjm* with a number of ad-hoc measures we thought could be related to the analyzed anti-patterns, namely TPC (Total Parameter Count), APC (Average Parameter Count), ATC (Abstract Type Count), VTC (Void Type Count), and EPM (Empty Parameters Methods).

We used the Spearman's rank correlation coefficient in order to establish the existing relations between the two kind of variables of our model, i.e. the OO metrics

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

[5] JasperReports, `http://jasperforge.org/projects/`

Table 2. Correlation between OO metrics and anti-patterns

Anti-patterns / Metrics	WMC	CBO	RFC	LCOM	LCOM3	LOC	DAM	CAM	AMC	TPC	APC	ATC	VTC	EPM
AP_1	**0.86**	0.42	0.52	0.36	-0.19	0.49	0.23	**-0.69**	0.11	**0.83**	0.38	0.25	0.33	0.33
AP_2	0.54	0.20	0.21	-0.07	-0.43	0.20	0.54	-0.48	-0.09	0.17	-0.29	0.19	0.33	**0.99**
AP_3	0.41	**0.98**	0.38	0.05	-0.16	0.33	0.14	**-0.65**	0.13	0.35	0.07	0.12	0.37	0.16
AP_4	**0.61**	0.38	0.34	0.0002	-0.31	0.32	0.23	-0.52	-0.01	0.59	0.26	0.12	0.45	0.39
AP_5	**0.79**	0.33	0.47	0.38	-0.15	0.44	0.24	-0.51	0.07	**0.71**	0.26	0.15	0.23	0.31
AP_6	0.50	0.35	0.31	-0.08	-0.35	0.27	0.21	-0.46	0.01	0.42	0.13	**0.60**	0.52	0.32

(independent variables) and the anti-patterns (dependent variables). Table 2 shows the correlation between the 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 cells values in bold are those coefficients which are statistically significant at the 5% level, i.e. p-value < 0.05, which is a common choice when performing statistical studies [18]. 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 fourteen metrics (i.e. WMC and CBO) are positively correlated to four of the six studied anti-patterns. Furthermore, there are three anti-patterns (*Ambiguous names*, *Enclosed data model* and *Redundant data models*) that are correlated to more than one OO metric. In this sense, in order to better clarify the analysis of the rationale behind the various high correlation factors, we selected the smallest sub-set of OO metrics that explain the six anti-patterns. We obtained two sub-sets, namely $< WMC, CBO, ATC, EPM >$ and $< WMC, CAM, ATC, EPM >$. Furthermore, we took the first sub-set as the CBO metric is more popular among developers and is better supported in IDE tools compared to the CAM metric. Moreover, 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. The results obtained from this correlation analysis show that the hypotheses defined in Section 3 are supported by our data, thus confirming their validity.

4.2 Early code refactorings for improving WSDL documents

The correlation among the WMC, CBO, ATC and EPM metrics and the anti-patterns, which were found to be statistically significant for the analyzed Web Service data-set suggest that, in practice, an increment/decrement of the metric values taken on the code of a Web Service directly affects anti-pattern occurrence 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. For the sake of

representativeness, we modified the services that presented all anti-patterns at the same time, which accounted for a 30% of the entire data-set.

In a first round of refactoring, we focused on reducing WMC by splitting the services having too much operations into two or more services so that on average the metric in the refactored services represented a 70% of the original value. Table 3 shows the impact on both WMC and its related anti-patterns, i.e. *Ambiguous names*, *Low cohesive operations in the same port-type* and *Redundant data models*. As depicted, on average, these two latter anti-patterns were reduced in 47.26% and 86.66%, respectively. This provides practical evidence to better support part of the correlation analysis of the previous section.

Table 3. Refactoring: Impact on WMC and its correlated anti-patterns

Metric and anti-patterns	**Original services (average)**	**Refactored services (average)**
WMC	19,32	4,00
Ambiguous names	42,08	42,08
Low cohesive operations in the same port-type	23,40	3,12
Redundant data models	114,00	60,12
Total number of anti-patterns	189,72	135,12

From Table 3, it can be seen that the performed refactoring introduced a significant increment of the average number of occurrences of the *Enclosed data model* anti-pattern while it did not affect the average number of occurrences of the *Ambiguous names* and *Empty messages* anti-patterns. The reasons behind these results are several limitations on the tool used to generate WSDLs, i.e. Java2WSDL. Despite these limitations, the total number of occurrences of anti-patterns was reduced in 30% on average.

In a second refactoring round, we focused on the ATC metric, which computes the number of parameters in a class that are declared as Object or data structures –i.e. collections– that do not use Java generics. In the latter case, when this practice is followed, these collections cannot be automatically mapped onto concrete XSD data-types for both the container 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 services obtained in the previous step in order to reduce ATC. Note that since ATC and WMC do not conflict between each other and at the same time are correlated to different anti-patterns, results are not affected by the order in which the associated refactorings are performed. Basically, the applied refactoring was to replace generic arguments with concrete ones.

By applying these modifications we were able to decrease the number of occurrences of the "Whatever types" anti-pattern. Note that the anti-pattern could not be removed completely as the ATC metric only operates at the service interface level. This means that if an interface parameter declared as a concrete data-type X has in turn in-

stance variables/generics with non-concrete data-types, the anti-pattern will nonetheless appear upon WSDL generation.

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 [19,20,11,7,8]. In particular, [8] 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 solutions. 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 [7], 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 [21,22,23].

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 [7]. 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 [8]. 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. Based on the

approach followed by several existing works in which some quality attributes of the resulting software are predicted during development time, 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 WSDLs. Interestingly, we found that these code refactorings effectively reduce anti-patterns, thus improving the resulting service contracts. Although most popular IDEs semi-automatically assist developers in applying these refactorings, we will automate them with the help of IntelliJ Idea[6], a Java-based IDE that has many built-in refactoring functions and is designed to be extensible.

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 will however 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.

Acknowledgments

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

References

1. Cristian Mateos, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Separation of concerns in Service-Oriented Applications based on pervasive design patterns. In *Proceedings of the 2010 Web Technology Track (WT) - ACM Symposium on Applied computing (SAC)*, pages 849–853. ACM Special Interest Group on Applied Computing, ACM, 2010.
2. John Erickson and Keng Siau. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of Database Management*, 19(3):42–54, 2008.
3. Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for Web Services. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *31th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, pages 372–383. Morgan Kaufmann, 2004.
4. Marco Crasso, Alejandro Zunino, and Marcelo Campo. Easy Web Service discovery: A Query-By-Example approach. *Science of Computer Programming*, 71(2):144–164, 2008.

[6] IntelliJ Idea, `http://www.jetbrains.com/idea`

5. Laura Rusu, Wenny Rahayu, and David Taniar. Intelligent dynamic XML documents clustering. In *22nd International Conference on Advanced Information Networking and Applications (AINA 2008)*, pages 449 –456. IEEE Computer Society, March 2008.
6. Marco Crasso, Alejandro Zunino, and Marcelo Campo. A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management*, 22(1):103–134, 2011.
7. Marco Crasso, Juan Manuel Rodriguez, Alejandro Zunino, and Marcelo Campo. Revising WSDL documents: Why and how. *IEEE Internet Computing*, 14(5):30–38, 2010.
8. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. 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, 2010.
9. Robert Van Engelen and Kyle Gallivan. The gSOAP toolkit for Web Services and peer-to-peer computing networks. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02)*, pages 128–135. IEEE Computer Society, 2002.
10. Shyam Chidamber and Chris Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
11. James Pasley. Avoid XML schema wildcards for Web Service interfaces. *IEEE Internet Computing*, 10:72–79, 2006.
12. Diomidis Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22:9–11, 2005.
13. Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Automatically detecting opportunities for web service descriptions improvement. In Wojciech Cellary and Elsa Estevez, editors, *Software Services for e-World*, IFIP Advances in Information and Communication Technology, pages 139–150. Springer, 2010.
14. Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 475–484. ACM Press, 2010.
15. Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 2nd edition, 1998.
16. Jagdish Bansiya and Carl Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.
17. K. L. Morris. Metrics for object-oriented software development environments. Master's thesis, M. I. T. Sloan School of Management, 1989.
18. Stephen Stigler. Fisher and the 5% level. *Chance*, 21:12–12, 2008.
19. Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32, 2005.
20. M. Brian Blake and Michael Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12:62–69, September 2008.
21. Ramanath Subramanyam and Mayuram Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
22. Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
23. Paulo Meirelles, Carlos Santos Jr., Joao Miranda, Fabio Kon, Antonio Terceiro, and Christina Chavez. A study of the relationships between source code metrics and attractiveness in free software projects. In *Brazilian Symposium on Software Engineering (SBES '10)*, volume 0, pages 11–20. IEEE Computer Society, 2010.