An Approach To Improve Code-first Web Services Discoverability at Development Time

José Luis Ordiales Coscia, Marco Crasso, Cristian Mateos and Alejandro Zunino
ISISTAN Research Institute - CONICET
UNICEN, Tandil, Argentina
[mcrasso|cmateos|azunino]@conicet.gov.ar

ABSTRACT

Previous efforts towards simplifying Web Service discovery have shown that avoiding some well-known WSDL specification anti-patterns yield quite good results in making more discoverable services. The anti-patterns, however, have been studied with contract-first Web Services, a service construction methodology that is much less popular in the software industry compared to code-first. We study a number of source code refactorings that can be applied at service development time to reduce the presence of anti-patterns in code-first WSDL documents. The cornerstone of these refactorings is a statistical correlation between common object-oriented (OO) metrics and the anti-patterns computed by using a data-set of real Web Services. We quantify the impact of the refactorings on Web Service discovery and show that more clear WSDL documents are generated and service discovery is greatly improved.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

Keywords

WEB SERVICES, CODE-FIRST, WSDL ANTI-PATTERNS, OO METRICS, WEB SERVICE DISCOVERY

1. INTRODUCTION

Web Service discovery plays a fundamental role in the development of service-oriented applications. Upon developing a system that invokes external Web Services, developers must first inspect service registries to discover and retrieve the service descriptions –WSDL documents– associated with the functionality they need to incorporate into the system. For the process of discovering and understanding Web Services from their WSDL documents to be effective, service providers must pay attention to the way WSDL documents are specified [13]. This is desirable as discoverable and understandable services potentially mean more client applications. For paid Web Services, this means more incomes.

SAC'12 March 26-30, 2012, Riva del Garda (Trento), Italy.

It has been shown that by considering a catalog of common anti-patterns upon specifying WSDL documents, service discoverability and clarity can be significantly improved [13]. A weak point of this approach as well as related efforts [3] is that services must be built in a contract-first manner [11], which means that WSDL interface specifications for services comes before service implementations. This practice is not very popular in the software industry as it requires upfront effort and many resources (e.g. training developers in XML Schema). Alternatively, with code-first, developers first implement a service and then generate its WSDL document by automatically deriving it from the implemented code via language-dependent tools like Axis Java2WSDL. However, this simple WSDL construction procedure is also counterproductive. Since developers do not have full control over their WSDL documents, anti-patterns can occur, which again places an unnecessary burden on service-oriented application developers.

In this paper we propose an approach that aims at avoiding the anti-patterns of [13] in code-first services. This approach bases on applying some simple refactorings in the code implementing services at development time. Refactorings are driven by some OO metrics that were found to have a significant statistical correlation with anti-patterns occurrence. In this sense, these metrics serve as early indicators that warn developers about the potential occurrence of WSDL anti-patterns. Since the refactorings are very simple and are independent of the WSDL generation tool, we believe our approach can be readily applicable in the industry. It is worth mentioning that, although our approach is in principle platform-independent, to limit the scope of our research, we focus on Web Services implemented in Java. Interestingly, Java is a language heavily employed for back-end Web development.

For evaluating the approach, we used a data-set of real public Web Services. We empirically analyzed the effects of applying our refactorings, which resulted in effective reduction of WSDL anti-patterns. In addition, we quantified the indirect effect of refactorings on service discovery by basing on this data-set and WS-QBE [5], an efficient Web Service registry infrastructure based on machine learning and text mining techniques. We have found that WSDL documents derived from service code that consider our refactorings are more discoverable than those that do not.

The rest of the paper is as follows. The next section explains the underpinnings of our approach. Section 3 reports a detailed experimental evaluation. Section 4 presents concluding remarks.

2. EARLY AVOIDING ANTI-PATTERNS

Most of the problems related to the efficiency of Web Service discovery engines stem from the fact that WSDL documents are poorly specified [13]. A poorly specified WSDL document is one that lacks proper comments, or contains non-representative, unre-

Anti-pattern	Occurs when
Ambiguous names	Non explanatory names are used for denoting 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 are not placed in separate XSD documents
Low cohesive operations in the same port-type	Port-types have weak semantic cohesion
Redundant data model	Different 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

Table 1: Main WSDL anti-patterns

lated or redundant keywords. These practices, besides negatively impacting on the retrieval effectiveness of service discovery engines, hinder humans' ability to reason about services functionality.

The study presented in [13] proposed the most comprehensive WSDL anti-patterns catalog to date. By exploring a large data-set of public WSDL documents, the authors abstracted away some recurrent practices that have been empirically proved as threats to WSDL discoverability and clarity. For brevity, the most harmful anti-patterns are listed in Table 1. Each identified anti-pattern has associated a sound refactoring action. The proposed associated solutions are based on refactoring actions for WSDL documents. Given a WSDL document having anti-patterns, the provider can methodically modify the document until all or some anti-patterns have been removed. As such, these solutions can be applied when services are developed in a contract-first manner only.

2.1 Approach and research hypotheses

To prevent WSDL documents from incurring in the anti-patterns presented in [13] when following the code-first method for building services, we established several hypotheses by using an exploratory approach to test the statistical correlation between OO metrics and the anti-patterns. Particularly, we hypothesized that either coupled classes, or weighted classes, or methods returning abstract or void types, are associated with higher number of anti-pattern occurrences within generated WSDL documents. The rationale behind our hypotheses is that typical code-first tools perform a mapping $T : C \rightarrow W$, where the service front-end class is represented as $C = [M[IR]]^*$, while $W = [O[IR]]^*$ stands for the generated WSDL document. The WSDL document W has a *port-type* for the service implementation class having as many *operations* O as public methods M are defined in class C. Each *operation* of W is associated with one input *message* I and one 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 [17] 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 because of the characteristics of these programming languages. Next we list the 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."* CBO [4] counts how many methods or instance variables defined by other classes are accessed by a given class. Code-first tools typically include in resulting WSDL documents as many XSD definitions as objects are exchanged by the methods of service classes. Then, increasing the number of external objects accessed by service classes increases the number of XSD data-type definitions.

Hypothesis 2 (H_2): *"The higher the number of public methods of a service class (WMC metric), the more frequent Low cohesive operations in the same port-type anti-pattern occurrences."* Since WMC [4] counts the methods of a class, a greater number of methods increases the probability that any pair of them are unrelated, i.e. having weak cohesion.

Hypothesis 3 (H_3): *"The higher the number of public methods of a service class (WMC metric), the more frequent the Redundant data models anti-pattern in the service."* The number of *message* elements within a WSDL document built under code-first tools, is equal to two times the number of *operation* elements. As each *message* may be associated with a data-type, then the number of redundant data-type definitions increases with the number of public methods, since this in turn increases the number of *operation* elements.

Hypothesis 4 (H_4): *"The higher the number of public methods of a service class (WMC metric), the more frequent the Ambiguous names anti-pattern occurrences."* Similarly to H_3, an increment in the number of methods lifts the number of non-representative names within a WSDL document, since for each method a 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 a service class that are declared as non-concrete data-types (ATC metric), the more frequent the Whatever types anti-pattern."* 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. Some code-first tools map abstract data-types and badly defined generics onto xsd:any constructors, which is the root cause for the *Whatever types* anti-pattern.

Hypothesis 6 (H_6): *"The higher the number of public methods of a service class that do not receive input parameters (EPM metric), the more frequent the Empty messages anti-pattern occurrences."* We designed the Empty Parameters Methods (EPM) metric to count the number of methods in a class that do not receive parameters. By increasing the number of methods without parameters, the number of *Empty messages* anti-pattern occurrences is increased, because code-first tools map this kind of methods onto operations associated with one input *message* element not conveying XML data.

The approach chosen for testing the hypotheses consists on gathering OO metrics from open source Web Services, looking for anti-patterns in their WSDL documents, and using correlation methods to validate the usefulness of these metrics for anti-pattern prediction. We gathered a data-set of around 90 different real services that contained, for each service, its implementation code and its WSDL document. Service implementations were collected via the Merobase crawler and the Exemplar engine [9]. Merobase allows users to harvest software components from sources such as Apache, SourceForge and Java.net. Complementary, we collected projects

Anti-pattern/Metric	WMC	CBO	ATC	EPM
Ambiguous names	**0.86** (H_4)	0.42	0.25	0.33
Empty messages	0.54	0.20	0.19	**0.99** (H_6)
Enclosed data model	0.41	**0.98** (H_1)	0.12	0.16
Low cohesive operations in the same port-type	**0.61** (H_2)	0.38	0.12	0.39
Redundant data models	**0.79** (H_3)	0.33	0.15	0.31
Whatever types	0.50	0.35	**0.60** (H_5)	0.32

Table 2: Anti-patterns and OO metrics: Correlation

from Google Code. Some of the retrieved projects actually implemented Web Services, whereas other projects contained coarse granular software components, which were "servified" to further enlarge the data-set. Overall, the data-set provided the means to perform a significant evaluation since it came from real developers.

We automated metrics recollection and anti-patterns detection, since the time needed to manually analyze each item of the data-set was 2 days/developer and it is an error prone task. For computing OO metrics, we extended *ckjm* [14], a Java-based tool that computes a sub-set of the Chidamber-Kemerer metrics [4].

For measuring the number of WSDL anti-patterns, we employed a detection tool [12] that automatically checks whether a WSDL document suffers from the anti-patterns of [13] or not. The tool uses heuristics to return a list of anti-patterns occurrences within a given WSDL document. Each heuristic is designed for detecting one anti-pattern, and reported experiments show that the averaged accuracy of the heuristics was 0.958 [12].

2.2 Empirical correlation between OO metrics and anti-patterns occurrence

The commonest way of analyzing the empirical relation between independent and dependent variables is by statistically testing experimental hypotheses [7]. We set the 6 anti-patterns described above as the dependent variables, and some OO metrics as the independent variables. We used the Spearman's rank correlation coefficient to establish the existing relations between the two kind of variables. Table 2 shows the correlation results. The values in bold are those coefficients that are statistically significant at the 5% level (p-value < 0.05), which is the usual configuration [15].

It can be seen from Table 2 that there is a statistically significant relation between the CBO metric and the number of occurrences of the *Enclosed data model* anti-pattern, with a correlation factor of 0.98 and a p-value of 0. We conclude that the hypothesis H_1 is supported by our data, thus accepting its validity. This occurs since code-first tools include in resulting WSDL documents as many XSD definitions as user defined objects are used by the service methods. Then, increasing the value of the CBO metric leads to a higher number of occurrences of the anti-pattern.

The hypothesis H_2 stated that the likelihood of non-cohesive operations increases with the number of public methods, which suggests a positive correlation between the WMC metric and the *Low cohesive operations in the same port-type* anti-pattern. As shown in Table 2, the correlation factor is the highest for this anti-pattern (0.61) and is also highly significant (p-value = 0). This allows us to accept the validity of the hypothesis H_2. The correlation between the two variables, on the other hand, tends to have an exponential nature. Let us consider the following example. Let S_1 be a Web Service with 3 unrelated methods M_1, M_2 and M_3. Therefore, WMC = 3 and *Low cohesive operations in the same port-type* = 3, since we would have the pair of non-cohesive operations $[M_1, M_2]$, $[M_1, M_3]$ and $[M_2, M_3]$. If we now add a fourth method M_4, the new values would be WMC = 4 and *Low cohesive operations in the same port-type* = 6. This is, as we increase the number of methods in the Web Service, the number of occurrences of the anti-pattern tends to increase exponentially.

The hypothesis H_3 stated that the probability of the *Redundant data models* anti-pattern increases with the number of public methods. The two variables present a highly significant (p-value = 0) strong positive correlation factor of 0.79 (see Table 2). Then, the hypothesis is supported by our data. Moreover, similarly to the relation between the WMC metric and the *Low cohesive operations in the same port-type* anti-pattern discussed before, the relation between the WMC metric and the *Redundant data models* anti-pattern has an exponential tendency. This arises from the way code-first tools generate WSDL documents. These tools define two *message* elements for each operation: one for its input parameters and one for its return type. As each *message* is associated with a data-type, the probability that any pair of methods share the same number and type of input parameters or the same return type increases exponentially with the number of public methods.

The hypothesis H_4 stated that the WMC metric was positively correlated with the *Ambiguous names* anti-pattern. From the correlation analysis shown in Table 2, it can be observed that there is a statistically significant relation between the two variables, with a correlation factor of 0.86 and a p-value of 0. This shows that the hypothesis H_4 is supported by our data, thus confirming its validity. As the value of the WMC metric increases, so does the number of operations and arguments, resulting in a higher probability that a sub-set of them use non-representative.

The hypothesis H_5 stated that an increment in the ATC metric may increase the likelihood of the *Whatever types* anti-pattern occurrences, which suggests a positive correlation between the two. As shown in Table 2, the two variables have a correlation factor of 0.60. This correlation is also highly significant, with a p-value of 0. It can be noted that this correlation factor is the highest for this anti-pattern. Therefore, we conclude that the validity of the hypothesis is confirmed from this data. The correlation between the metric and the anti-pattern stems from the use of generics and abstract types in the service code. For example, let us suppose a service with a single operation that receives a List and a String as input parameters and returns a Hashmap as output parameter. The generated WSDL document using the Java2WSDL tool would map both the List type and the Hashmap type onto <anyType> constructors, resulting in two occurrences of the *Whatever types* anti-pattern.

The hypothesis H_6 stated that a greater number of methods without input parameters increases the probability of the *Empty messages* anti-pattern, suggesting a positive correlation between the EPM metric and the anti-pattern. From Table 2 it can be seen that this is the case, as shown by the highly statistically significant relation between the two variables (correlation factor of 0.99 with p-value = 0). Then, we conclude that the hypothesis is supported by our data. The high correlation factor between the two variables is, once again, due to the way code-first tools generate WSDL documents. For those methods that do not receive any parameters, tools still generate an operation element associated with one empty input message element that is not intended to transport any XML data.

3. EXPERIMENTAL RESULTS

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,

Metric or anti-pattern	Decrement
WMC	79.29%
Ambiguous names	0.00%
Low cohesive operations in the same port-type	86.66%
Redundant data models	47.26%

Table 3: First round of refactorings: Impact on WMC and its correlated anti-patterns

Metric or anti-pattern	Decrement
WMC	79.29%
ATC	100.00%
Low cohesive operations in the same port-type	13.33%
Redundant data models	52.73%
Whatever types	21.09%

Table 4: Second round of refactoring: Impact on WMC and ATC and their correlated anti-patterns

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. To quantify these effects, we carried out some source code refactorings driven by these metrics on the data-set (Section 3.1). Then, we measured the impact of performing such refactorings on service discovery (Section 3.2).

3.1 Effects of early code refactorings on anti-patterns occurrence

For representativeness, we modified the services that presented all anti-patterns at the same time, which accounted for a 30% of the data-set. In a first round, we focused on reducing WMC by employing the Fowler et al.'s MOVE METHOD refactoring [8], i.e. splitting the services having too many 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: *Ambiguous names*, *Low cohesive operations in the same port-type* and *Redundant data models*. On average, the two latter anti-patterns were reduced in 47.26% and 86.66%, respectively. *Ambiguous names* anti-pattern occurrences were the same, because the refactoring was driven by WMC and thus method and parameter names were not modified.

Unfortunately, the refactorings introduced a significant increase of the average number of occurrences of the *Enclosed data model* anti-pattern. The original Web Services had on average 6.84 occurrences versus the 20.56 average occurrences in the refactored services, which is due to a limitation regarding complex data-type reuse of the current implementation of Java2WSDL, the tool used to generate WSDLs. For example, a service having 10 operations whose signatures use the same class definition *C* produces only one occurrence of the anti-pattern. But, after refactoring, if the service is divided for example into 5 new services with 2 operations each, the number of occurrences increases to five since we have 5 services with one occurrence each. This is, Java2WSDL has no "global" sense of data-type models upon WSDL generation. Nevertheless, this does not translate into an irremediable problem since an alternative code refactoring to avoid this situation, which in fact might reduce CBO, is to replace one or more user-provided classes within a Web Service implementation with native data-types. This practice would however produce a less precise and expressive class (and potentially data) model, which attempts against the legibility and clarity of the exposed data-types of services. Then, the desired data-type duplicity/legiblity balance should be established.

Finally, the fall in the occurrences of the *Redundant data models* anti-pattern after the refactoring is also due to the lack of sense of global data-types, but of the anti-pattern detector, in this case. If two services define the same data-type, the detector will not count it as an anti-pattern occurrence. Instead, if a service has 2 operations both using the defined data-type twice, the detector counts 2 anti-pattern occurrences. However, after the refactoring, if the service is divided in 2 new services with one operation having the same data-type each, the detector does not count the anti-pattern.

In a second round of refactorings, we focused on the ATC metric, which computes the number of parameters in a class declared as Object or collections that do not use Java generics. Usually, collections cannot be automatically mapped onto concrete XSD data-types by the generation tool for both the container and the contained data-type in the final WSDL. A similar problem arises with parameters whose data-type is Object. Hence, we modified the services obtained in the previous step to reduce ATC. Since ATC and WMC are correlated to different anti-patterns, results are not affected by the order in which the associated refactorings were performed.

The applied refactorings consisted in replacing arguments declared as Object with a concrete data-type whenever possible. Also, instead of replacing parameters declared as Vector, List, etc., with their generic counterparts, which is known as INTRODUCE TYPE PARAMETER [8], we decided to replace the former with array structures. Overall, by applying these modifications decreased 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 instance variables/generics with non-concrete data-types, the anti-pattern will nonetheless appear upon WSDL generation.

Finally, the *Empty messages* anti-pattern, which is associated with the EPM metric, could not be removed since the anti-pattern is caused by the way Java2WSDL builds WSDL messages. Unlike WMC, ATC and to a lesser extent CBO, taking EPM into account has to be completely done at the WSDL generation level. This concretely means that the generation tool should not build an empty input message for class methods without parameters. Table 4 shows the metrics and anti-patterns that were reduced after refactoring.

3.2 Effects of early code refactorings on service discovery

We measured the implications on discovery of early detecting anti-patterns in service implementations via code refactorings to avoid anti-patterns from target associated WSDL documents. In other words, we analyzed whether placing effort on refactoring service implementations actually rewards developers by improving their services chances of being discovered or not.

The evaluation consisted of three steps. In a first step, code-first WSDL documents were grouped into two categories. One category called "Refactored" consisted of those WSDL documents that were generated after applying the proposed refactorings to a sub-set of the service implementations gathered from open source projects. Another category ("Original") had the original versions of the improved WSDL documents. Second, we supplied a service registry with both categories of WSDL documents. Third, we queried the employed registry using one query per available service operation in the published services. For each query we employed

the Precision-at-*n* metric to measure the position at either the original or the refactored WSDL documents were retrieved. Precision-at-*n* computes precision at different cut-off points. For example, if the top 5 documents are all relevant to a query and the next 5 are all non-relevant, we have a precision of 100% at a cut-off of 5 documents but a precision of 50% at a cut-off of 10 documents. Finally, we averaged the results over the total number of queries.

The refactored documents were WSDL files whose implementations were modified to consider not some, but all the refactorings discussed previously. The reason behind this decision was that, even when some anti-patterns affect service discovery more than others [13], they all negatively impact on WSDL discoverability. Hence, the best results in terms of retrieval effectiveness are obtained when removing all the anti-patterns, which means that all the refactorings associated with correlated OO metrics have to be considered. In practice, modifying service implementations by taking into account all the refactorings is not an expensive task since most of them can be easily performed with the help of modern IDEs.

For the experiments, we used a publicly available registry implementation of the approach to service discovery presented in [5]. This registry exploits relevant information contained in WSDL documents by combining text-mining and machine learning techniques to remove redundant plus non-relevant data and build a vectorial representation of each service, respectively. This is a classical model borrowed from the Information Retrieval area, known as Vector Space Model [5], which is widely employed for discovering services, as reported in a recent survey of approaches to discover services [6]. With this model, documents are seen as collection of terms, whereas each dimension of the space corresponds to a separate term, thus documents having similar contents are represented as vectors located near in the space. Accordingly, searching related documents translates into searching nearest vectors. Any form of textual based queries, ranging from single keywords to textual descriptions of their needs, are mapped onto a vector, and in turn those vectors that are near to the query vector are marked as relevant [5].

For fairness we built the employed queries from the source code of original service implementations. This assumption is analogous to the Query-By-Example concept presented in [5], upon which the registry is built. For example, the query for looking for operations whose signature is: "getActiveWorkflows(userID:string)" may be "get active workflows". In fact, the employed registry splits combined words within queries. Following this assumption, 463 queries were built, one per offered operation. We associated two WSDL documents with each query, one document belonging to the Original service category, while another from the Refactored one. For the association we arbitrarily selected the WSDL documents containing the operation needed.

The Precision-at-*n* results were computed for each query with *n* in [1,10]. This window size was chosen because we want a good balance between the number of candidates and the number of relevant candidates retrieved. Moreover, we believe that a developer can easily examine 10 Web Service descriptions in a usual discovery scenario. Therefore, by setting $n = 10$, the considered number of relevant services in the result list is up to 10 candidates. Figure 1 depicts the averaged Precision-at-*n* results for the 463 queries by smoothing these results using Bézier curves. Results show that Refactored WSDL documents were ranked before their Original counterparts. Having a higher Precision-at-*1* means that a relevant service was retrieved at the top of the result list. Precision-at-*1* was 66.1% and 2.6% for Refactored and Original categories, respectively. Then, the WSDL documents associated with services whose implementations had been refactored, were ranked first in the 66.1% of the cases. As emphasized in Figure 1, 93% of relevant Refactored documents were retrieved at the forth position, in the worst case, whereas a fraction of 54% of relevant Original WSDL documents were retrieved at the same position. Accordingly, discoverers would have to analyze up to only 4 candidates until finding a relevant service when employing Refactored WSDL documents in 93% of the cases. Clearly, Refactored WSDL documents were better ranked compared to Original ones. These results have a great impact on discoverability because users tend to select higher ranked search results [1]. For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the next one is, at most, 60% [1].

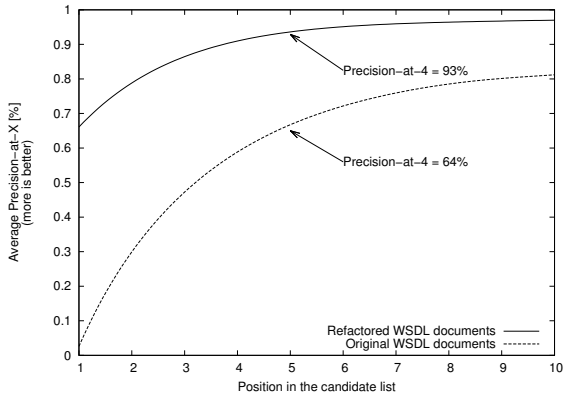

Figure 1: Averaged Precision-at-n results.

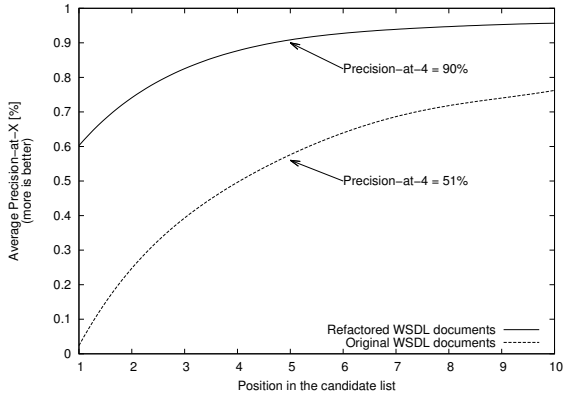

Figure 2: Averaged Precision-at-n when also considering the WSDL documents gathered by Heß et al [10].

The fact that only Original and Refactored versions of the same WSDL documents coexist in a registry undoubtedly is unrealistic, though it is useful for comparison purposes. We have assessed the implications of applying the proposed refactorings in a more realistic scenario, by reproducing the same experiment with the data-set of ca. 400 publicly available WSDL documents described in [10]. We published this data-set in our registry along with the Original and Refactored groups of WSDL documents. Therefore, this experiment was equal to the former except for the second step, which has been modified to simulate a real world scenario.

Figure 2 depicts the averaged 463 Precision-at-*n* results. The tendency of ranking Refactored WSDL documents first remained the same, though Precision-at-*n* results fell by 2.85% and 7.02% for the Refactored and Original categories, respectively. This indicates that the discoverability of Original WSDL documents was

more affected by the noise introduced in the registry than the discoverability of the Refactored ones. Figure 2 shows, for instance, that 51% of the Original WSDL documents were ranked at the fourth position at most, whereas for the first experiment this value was 64%, i.e. a fall of 13 %. Regarding Refactored WSDL documents, the Precision-at-*4* results fell only 3%. These results may indicate that though more WSDL documents have been published in the registry, Refactored ones are still more discoverable.

Both experiments provide empirical evidence showing that employing the proposed refactorings on service implementations improve the discoverability of their WSDL documents. Due to the approach to service discovery used, Precision-at-*n* results are data-set and query-set specific, and cannot be generalized to other experimental conditions. However, as the proposed refactorings rely on re-grouping operations for improving service internal cohesion, remodeling data-types for making them representative of domain objects, and following good naming conventions for making operations and arguments names self-descriptive, it is reasonable to expect some retrieval advantage when applying the refactorings, versus not applying them. This is because the efficiency of most approaches to Web Service discovery depends on the descriptiveness of published WSDL documents [13].

4. CONCLUSIONS AND FUTURE WORK

WSDL documents play a crucial role in making Web Services understandable and discoverable. Several WSDL specification bad practices or WSDL anti-patterns have been documented in the literature, which actually attempt against this goal. Therefore, the problem of how to correct, or even better, avoid such anti-patterns when deriving WSDL documents is of major importance.

We have described an approach to minimize anti-patterns occurrence when using code-first. This is achieved by considering the values of some OO metrics taken at the source code implementing services which were found to be statistically correlated to WSDL anti-patterns. By using a data-set of real Web Services, we investigated the effect of applying some metric-driven code refactorings to the Web Services on the anti-patterns in the generated WSDLs. We found that these refactorings not only improve WSDL clarity but also allow for more efficient service discovery.

We are extending our work in several directions to generalize our results. First, we will incorporate into our analysis other WSDL generation tools, such as EasyWSDL and JBoss' *wsprovide*. Second, the relationships between the anti-patterns and other OO metrics, e.g. traditional metrics [16] or newer ones [2], could in turn lead to explore the effects of others early refactorings on anti-patterns and service discovery. Precisely, a third research line involves performing experiments with other Web Service discovery engines apart from WS-QBE. Lastly, as part of an independent project, an l-commerce system that comprises around 40 code-first Web Services is being developed. We will eventually consider these Web Services to get a larger experimental date-set.

Acknowledgments

We acknowledge the financial support provided by ANPCyT through grant PAE-PICT 2007-02311.

References

[1] E. Agichtein, E. Brill, S. Dumais, and R. Ragno. Learning user interaction models for predicting Web search result preferences. In *29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–10. ACM, 2006.

[2] J. Al Dallal. Measuring the discriminative power of object-oriented class cohesion metrics. *IEEE Transactions on Software Engineering*, 2010. To appear.

[3] M. Blake and M. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12:62–69, September 2008.

[4] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] M. Crasso, A. Zunino, and M. Campo. Query by example for Web Services. In *2008 Web Technology Track (WT) - ACM Symposium on Applied computing (SAC)*, pages 2376–2380. ACM, 2008.

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

[7] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

[9] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *32nd ACM/IEEE International Conference on Software Engineering*, pages 475–484. ACM, 2010.

[10] A. Heß, E. Johnston, and N. Kushmerick. ASSAM: A tool for semi-automatically annotating semantic Web Services. In *International Semantic Web Conference*, volume 3298 of *LNCS*, pages 320–334. Springer, 2004.

[11] C. Mateos, M. Crasso, A. Zunino, and M. Campo. Separation of concerns in Service-Oriented Applications based on pervasive design patterns. In *2010 Web Technology Track (WT) - ACM Symposium on Applied computing (SAC)*, pages 849–853. ACM, 2010.

[12] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo. Automatically detecting opportunities for Web Service descriptions improvement. In *Software Services for e-World*, IFIP Advances in Information and Communication Technology, pages 139–150. Springer, 2010.

[13] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.

[14] D. Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22:9–11, 2005.

[15] S. Stigler. Fisher and the 5% level. *Chance*, 21:12–12, 2008.

[16] F. Tsui and O. Karam. *Essentials of Software Engineering*. Prentice Hall, 2006.

[17] R. Van Engelen and K. Gallivan. The gsoap toolkit for Web Services and Peer-to-Peer computing networks. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 128–135. IEEE, 2002.