# Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools

**José Luis Ordiales Coscia**
E-mail: jlordiales@gmail.com

**Cristian Mateos***

ISISTAN-CONICET - UNICEN University
Tandil (B7001BBO), Buenos Aires, Argentina.
Tel./Fax: +54 (249) 443-9682 ext. 35/443-9681
E-mail: cmateos@conicet.gov.ar
Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
*Corresponding author

**Marco Crasso**

ISISTAN-CONICET - UNICEN University
E-mail: mcrasso@conicet.gov.ar
Also CONICET

**Alejandro Zunino**

ISISTAN-CONICET - UNICEN University
E-mail: azunino@conicet.gov.ar
Also CONICET

**Abstract:** Service-Oriented Computing (SOC) is a recent paradigm that promotes building applications as a set of coarse-grained, remote software components called services. SOC just represents a paradigm and as such it must be materialized through specific technologies, being Web Services the most common choice. In Web Service terms, each service is composed of an implementation and an abstract description of its functionality 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. Even when this practice is cheaper, bad design and coding practices already present in a service implementation may end up affecting the quality of the resulting WSDL document. For Web Services to be reusable, good WSDL designs are crucial. In a previous paper Mateos et al. (2011) it was shown that there is a high correlation between established Object-Oriented (OO) metrics from the source code implementing services and the occurrences of "anti-patterns" in WSDL documents. In this follow-up paper, these previous results are extended to all the existing WSDL generation tools and perform a detailed analysis of the impact of individual early source code OO metric-driven refactorings on the quality of the obtained WSDL documents. The generalized nature of the experiments makes the findings readily applicable in the industry.

**Keywords:** SERVICE-ORIENTED COMPUTING; WEB SERVICES; CODE-FIRST; OBJECT-ORIENTED METRICS; WSDL ANTI-PATTERNS; WSDL GENERATION TOOLS; JAVA.

**Reference** to this paper should be made as follows: Ordiales Coscia, J. L., Mateos, C., Crasso, M. and Zunino, A. (xxxx) 'Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools', *Int. J. Web and Grid Services*, Vol. x, No. x, pp.xxx–xxx.

**Biographical notes:** José Luis Ordiales Coscia is an MSc. candidate at the UNICEN, working under the supervision of Cristian Mateos and Marco Crasso. His thesis is about early improving understandability and discoverability of code-first Web Services.

Cristian Mateos http://www.exa.unicen.edu.ar/~cmateos received a Ph.D. degree in Computer Science from the UNICEN, in 2008, and his M.Sc. in Systems Engineering in 2005. He is a full time Teacher Assistant at the UNICEN and member of the ISISTAN and the CONICET. He is interested in parallel/distributed programming, Grid middlewares and Service-oriented Computing.

Marco Crasso http://www.exa.unicen.edu.ar/~mcrasso received a Ph.D. degree in Computer Science from the UNICEN in 2010. He is a member of the ISISTAN and the CONICET. His research interests include Web Service discovery and programming models for SOC.

Alejandro Zunino http://www.exa.unicen.edu.ar/~azunino received a Ph.D. degree in Computer Science from the UNICEN, in 2003, and his M.Sc. in Systems Engineering in 2000. He is a full Adjunct Professor at UNICEN and member of the ISISTAN and the CONICET. His research areas are Grid computing, Service-oriented computing, Semantic Web Services and mobile agents.

## 1 Introduction

The success encountered by the Internet encourages practitioners, companies and governments to create software that uses information and services that third-parties have made public in the Web. Service-Oriented Computing (SOC) is a relatively new computing paradigm that supports the development of distributed applications in heterogeneous environments Erickson and Siau (2008) and has radically changed the way applications are architected, designed and implemented Mateos et al. (2010). 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 WWW-based way for implementing the SOC paradigm. Web Services are *services* with clear interfaces that can be published, located and consumed through ubiquitous Web protocols Erickson and Siau (2008)

such as SOAP W3C Consortium (2007). When employing Web Services, a provider describes each service technical contract, a.k.a. its interface, in WSDL, an XML-based language designed for specifying service functionality as a set of abstract operations with inputs and outputs, and to associate binding information so that consumers can invoke the offered operations. These interactions among a service producer, a registry and a consumer are shown in Figure 1.

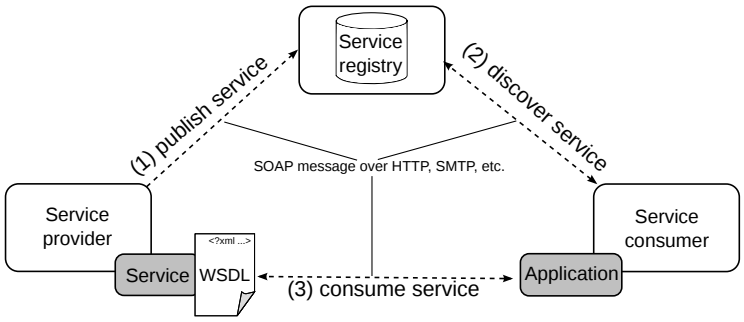



**Figure 1**: The Web Services model (extracted from Mateos et al. (2011)).

To make their WSDL documents publicly available, providers used to 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. 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. In recent years several syntactic Web Service registries such as Woogle Dong et al. (2004), WSQBE Crasso et al. (2008) and seekda![1] have however emerged and took over UDDI. These registries work by applying text mining or machine learning techniques, such as XML supervised classification Crasso et al. (2008) or clustering Rusu et al. (2008), to improve the retrieval effectiveness of the same keyword-based discovery process Crasso et al. (2011). In the future, it is expected that WSDL documents will be automatically enhanced with semantical annotations, i.e., a link from a specific WSDL part to a concept of an ontology, but a requirement to reach such an ambitious goal is that WSDL descriptions should be self-explanatory Crasso et al. (2010b).

All in all, service contract design plays one of the most important roles in enabling third-party consumers to understand, discover and reuse services Crasso et al. (2010a). On one hand, unless appropriately specified by its provider, a service contract can be counterproductive and obscure the purpose of a service and 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 Rodriguez et al. (2010a). Moreover, a WSDL description without much comments of its operations can make the associated Web Service difficult to be discovered, and particularly discovery precision of syntactic registries is harmed when dealing with poorly described WSDL documents Rodriguez et al. (2010a).

In Rodriguez et al. (2010a), common discoverability bad practices, or *anti-patterns* for short, found in public WSDL documents are studied. The same authors in Rodriguez et al. (2010c) provide a set of guidelines service providers should take into account to obtain clear, discoverable service contracts. However, a requirement inherent to applying these guidelines is that services are built in a *contract-first* manner, which means first deriving the WSDL contract of a service and then supplying an implementation for it. However, the most used approach to build Web Services by the industry is *code-first*, which means first implementing a service and then generating the WSDL contract by automatically deriving this latter from the implemented code. In this way, WSDL documents are not created manually but are automatically derived via WSDL generation tools. Consequently, anti-patterns may occur in the resulting WSDL documents when bad implementation practices are followed Crasso et al. (2010a).

In Mateos et al. (2011) it was shown that there is a statistical significant, high correlation between several traditional and ad-hoc Object-Oriented (OO) metrics and the anti-patterns. Particularly, that work studied the feasibility of avoiding these anti-patterns by using 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 in the Web Service implementation phase prior to WSDL generation. However, one of the main limitations of the mentioned work is that only one WSDL generation tool (i.e., Java2WSDL) was used for the experimental analysis. Therefore, the results presented a high dependency with the tool chosen and could not be generalized.

In this paper, the analysis presented in Mateos et al. (2011) was extended to include extra Java-based tools for the generation of the Web Services contracts including WSProvide, Java2WS and EasyWSDL, which brings the findings to a broader audience. This approach benefits many software practitioners in the industry, where code-first service construction is commonplace. Specifically, through some statistical analysis, this paper shows that a small sub-set of the OO metrics studied is highly correlated to the studied anti-patterns and, more important, these results hold valid for the most of the employed WSDL generation tools. Based on this, the present paper analyzes several simple code refactorings that developers can use to avoid anti-patterns in their service contracts. Additionally, this paper quantifies the impact of applying both each refactoring individually or in tandem in the number of anti-patterns present in resulting WSDLs. For the experiments, unlike Mateos et al. (2011), in which a data-set of 90 Web Services was used, an extended data-set of 154 Web Services was employed. It is worth noting that this is to date the largest code-first Web Services data-set existing in the literature.

The rest of the paper is structured as follows. Section 2 gives some quick background on the WSDL anti-patterns and code-first tools for the Java language. 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. For the sake of readability and self-containment, some of the explanations reported in Mateos et al. (2011) have been deliberately included.

## 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 consumers, while the latter part specifies technological aspects, such as transport protocols and network addresses. Consumers use the functional descriptions to match third-party services to their needs, and the technological details to invoke the selected service.

With WSDL, service functionality is described as one or more *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 as 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.

A WSDL document is intended to be the only publicly available software artifact describing a Web Service. Thus, many approaches to Web Service discovery are based on WSDL service descriptions Crasso et al. (2011). 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 Crasso et al. (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, poorly written WSDL documents deteriorate approaches retrieval effectiveness.

The work published in Rodriguez et al. (2010a) introduces the WSDL discoverability anti-patterns (see Table 1). This work measures the impact of WSDL anti-patterns 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 Rodriguez et al. (2010c).

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 contract of a service and then supply an implementation for it. Contract-first however is not very popular among developers because it requires a bigger effort than *code-first*. Code-first promotes first implementing a service and then generating the corresponding service contract by automatically extracting and deriving the interface from the implemented code. In general, code-first tools map the implementation code $C$ to a WSDL document $W$, formally $T : C \rightarrow W$.

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

| 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 do not 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 ($AP_5$) | Many data-types for representing the same objects of the problem domain. |
| Whatever types ($AP_6$) | A special data-type is used for representing any object of the problem domain. |

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 WSDL.exe, Java2WSDL, and gSOAP Van Engelen and Gallivan (2002) 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. The same applies to tools for the same language, such as the case of the four Java-based tools used in this paper.

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. 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.

It is known that it is possible to avoid WSDL anti-patterns *early* in the implementation phase by basing on classic API metrics taken at service implementations Mateos et al. (2011). As explained in Crasso et al. (2010a), the anti-patterns are associated with API design qualitative attributes, in the sense that some anti-patterns spring when well-established API design golden rules are not applied. 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 approach underlying the present paper

(i.e., Mateos et al. (2011)) avoids WSDL discoverability anti-patterns previous to build WSDL documents from service implementations.

## 3 Hypothesis statements for early WSDL anti-patterns detection

The approach in Mateos et al. (2011) aims at allowing providers to prevent their WSDL documents from incurring in the WSDL anti-patterns presented in Rodriguez et al. (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 $T$. The second fact underpinning the approach is that anti-patterns are associated with API design attributes Crasso et al. (2010a), which can be measured by using Object-Oriented (OO) class-level metrics. A well-known metric catalog in this line is the Chindamber and Kemerer's catalog Chidamber and 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"* measures WSDL *operations* cohesion. Then, by employing well-known software engineering metrics on a service code $C$, a provider can estimate how the resulting WSDL document $W$ will be like in terms of anti-pattern occurrences Mateos et al. (2011), since a known mapping $T$ relates $C$ with $W$.

Based on these facts, several hypotheses by using an exploratory approach to test the statistical correlation among OO metrics and the anti-patterns can be established. The hypotheses that actually hold are shown below:

**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 and 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. Then, 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 and Kemerer (1994) metric counts the methods of a class. Therefore, 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, 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 $H_3$, 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 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 are root causes for the *Whatever types* anti-pattern Pasley (2006), Rodriguez et al. (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. EPM (Empty Parameters Methods) counts the number of methods in a class that do not receive parameters. Then, 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 under the four Java-based WSDL generation tools considered.

## 4 Statistical analysis and experiments

The hypotheses of the previous section were tested by 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. This allowed us to assess the usefulness of the metrics for anti-pattern prediction. To perform the analysis, we implemented a software pipeline including software tools for automating metrics recollection, WSDL document generation, and finally anti-patterns detection. For

the first task, we extended *ckjm* Spinellis (2005), a Java-based tool that computes the Chidamber-Kemerer metrics Chidamber and Kemerer (1994), whereas for the second part we used the already mentioned four Java-based WSDL generation tools.

To measure anti-patterns, we employed a WSDL anti-pattern detection tool Rodriguez et al. (2010b). The WSDL Anti-patterns Detector Rodriguez et al. (2010b) (or Detector) automatically checks whether a WSDL document suffers from the anti-patterns of Rodriguez et al. (2010a) or not based on a given WSDL document as input. The Detector includes heuristics to deal with 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, and heuristics to deal with detecting *Ambiguous names* and *Low cohesive operations in the same port-type* anti-patterns, which require not only structural but also textual analysis of WSDL documents.

In the tests, a data-set of 154 different real services was used, whose implementations were collected by using the Merobase component finder (`http://merobase.com`) and the Exemplar code search engine Grechanik et al. (2010). Merobase allows users to harvest software components from a several sources (e.g., Apache, SourceForge, and Java.net) and supports interface-driven and text-driven searches. Exemplar relies on a hybrid approach to keyword-based search that combines textual processing and intrinsic qualities of code to mine repositories. Complementary, projects from Google Code were collected, plus around 60 Web Services from a local software company that most of its applications are designed following the SOC paradigm. All in all, the data-set included Web Service implementations from real-life software engineers. After building the data-set, the associated services were uniformized by explicitly providing a Java interface in order to facade their implementations. Then, 4 WSDL documents were obtained by feeding four different WSDL generation tools with the corresponding interface, namely WSProvide[2], Java2WS[3], EasyWSDL[4] and Java2WSDL[5].

The rest of the Section is structured as follows. Section 4.1 describes the statistical correlation analysis between OO metrics and anti-patterns performed on the above data-set. Section 4.2 explores several service refactorings at the source code level and their effect on the anti-patterns of WSDL documents. The Section also summarizes the main findings that are of interest to Web Service practitioners.

### *4.1 OO metrics and WSDL anti-patterns: Correlation analysis*

Similar to Mateos et al. (2011), in this study the 6 anti-patterns described up to now were set as the dependent variables, whose values were produced by using the Detector, while OO metrics were set as the independent variables, which were computed via the *ckjm* tool. As independent variables, 11 OO metrics were used. Specifically, WMC, CBO, RFC, and LCOM have been selected from the work of Chindamber and Kemerer Chidamber and 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

**Table 2** Descriptive statistics for anti-patterns.

| Probabilistic metric | Tool | Ambiguous names | Empty messages | Enclosed data model | Low cohesive... | Redundant data models | Whatever types |
|---|---|---|---|---|---|---|---|
| Minimum | WSProvide | 2.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Java2WS | 2.00 | 0.00 | 2.00 | 0.00 | 0.00 | 0.00 |
| | EasyWSDL | 0.00 | 0.00 | 2.00 | 0.00 | 0.00 | 0.0 |
| | Java2WSDL | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.0 |
| Maximum | WSProvide | 316.00 | 0.00 | 0.00 | 1263.00 | 0.00 | 0.00 |
| | Java2WS | 316.00 | 0.00 | 227.00 | 1288.00 | 1073.00 | 15.00 |
| | EasyWSDL | 433.00 | 0.00 | 225.00 | 1554.00 | 1142.00 | 17.00 |
| | Java2WSDL | 243.00 | 11.00 | 44.00 | 910.00 | 891.00 | 17.00 |
| Mean | WSProvide | 15.58 | 0.00 | 0.00 | 12.28 | 0.00 | 0.00 |
| | Java2WS | 15.57 | 0.00 | 46.66 | 12.49 | 39.43 | 0.64 |
| | EasyWSDL | 17.94 | 0.00 | 17.29 | 12.72 | 29.24 | 0.33 |
| | Java2WSDL | 13.43 | 0.57 | 5.41 | 9.78 | 23.46 | 0.92 |
| Std. Deviation | WSProvide | 36.91 | 0.00 | 0.00 | 102.99 | 0.00 | 0.00 |
| | Java2WS | 36.92 | 0.00 | 43.35 | 104.98 | 140.06 | 1.42 |
| | EasyWSDL | 46.57 | 0.00 | 25.68 | 125.41 | 129.47 | 1.66 |
| | Java2WSDL | 28.77 | 1.64 | 7.09 | 74.99 | 100.46 | 1.86 |

methods. In addition, the CAM (Cohesion Among Methods of Class) metric from the work of Bansiya and Davis Bansiya and Davis (2002) was picked. CAM computes the relatedness among methods based upon the parameter list of these methods. Additionally, a number of extra metrics that could be related to the WSDL metrics were used, 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 metrics studied are shown in Table 3 while the same information for 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* ($AP_6$), considering each generation tool is shown in Table 2. Several interesting facts can be observed from this data. First of all, the experimental data show that when using the WSProvide tool no occurrences of $AP_2$, $AP_3$, $AP_5$ and $AP_6$ were detected. This is due to the fact that this particular WSDL tool is the only one that defines every XSD type on a separate file and thus, they are not taken into consideration by the Detector. However, even when the tool is unable to detect the occurrences of these anti-patterns they are still on the XSD file. Therefore, their negative effect on human discoverers' ability to understand and select the service remains Rodriguez et al. (2010c).

**Table 3** Descriptive statistics for OO metrics.

| OO metric / Probabilistic metric | Minimum | Maximum | Mean | Std. Deviation |
|---|---|---|---|---|
| WMC | 1.00 | 97.0 | 5.73 | 11.13 |
| CBO | 0.00 | 27.0 | 2.02 | 2.91 |
| RFC | 1.00 | 97.0 | 5.73 | 11.13 |
| LCOM | 0.00 | 4656.0 | 75.21 | 427.42 |
| LOC | 1.00 | 97.0 | 5.73 | 11.13 |
| CAM | 0.13 | 1.0 | 0.78 | 0.23 |
| TPC | 0.00 | 228.0 | 10.91 | 24.23 |
| APC | 0.00 | 17.0 | 2.04 | 1.83 |
| ATC | 0.00 | 20.0 | 1.09 | 2.25 |
| VTC | 0.00 | 25.0 | 1.05 | 3.53 |
| EPM | 0.00 | 11.0 | 0.57 | 1.64 |

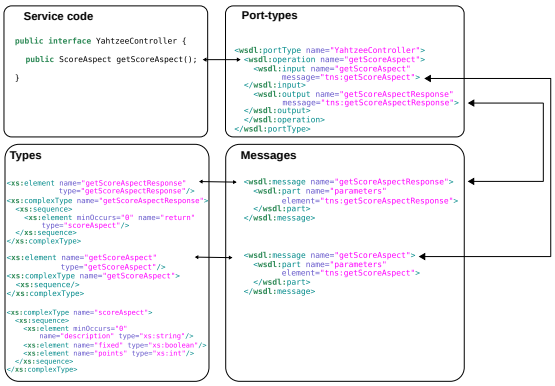



**Figure 2**: Simple WSDL generation for Java2WS and EasyWSDL.

It is also worth noting that both EasyWSDL and Java2WS present no occurrences of $AP_2$. To understand the reasons behind this result consider the WSDL generation example in Figure 2. It can be seen that when generating an operation with no input parameters, although the corresponding message is not empty, now the data-type referenced from this message is. Therefore, even when technically there are no empty messages, the anti-pattern was not solved but just pushed one step down the generation process, which is not spotted by the Detector.

The Spearman's rank correlation coefficient was used in order to establish the existing relations between the dependent and independent variables of the above statistical model. Table 4 depicts the correlation factors among the studied OO metrics. 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

**Table 4** Correlation among OO metrics.

| OO metric | WMC | CBO | RFC | LCOM | LOC | CAM | TPC | APC | ATC | 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** |
| ATC | - | - | - | - | - | - | - | - | **1.00** | 0.03 | **-0.18** |
| VTC | - | - | - | - | - | - | - | - | - | **1.00** | **0.38** |
| EPM | - | - | - | - | - | - | - | - | - | - | **1.00** |

studies Stigler (2008). The sign of the correlation coefficients defines the direction of the relationship, i.e., positive or negative. A positive relation means that when the independent 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 dependent variables grow, the independent 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 4 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 studied data-set, it can be seen from Table 4 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.

Tables 5a, 5b, 5c and 5d show the correlation between the OO metrics and the anti-patterns for WSProvide, Java2WS, EasyWSDL and Java2WSDL, respectively. The factors in bold represent those with a p-value ¡ 0.05. Additionally, for the sake of readability, a graphical approach to depict the correlation matrixes of Tables 5a, 5b, 5d and 5c was used, which is shown in Figure 3. 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 in Figure 3 that the six bigger circles, i.e., the six highest statistically significant correlation factors, correspond precisely with the six defined hypotheses.

**Table 5** Correlation between OO metrics and WSDL anti-patterns using different code-first tools.

(a) Correlation between OO metrics and anti-patterns for WSProvide.

| Anti-pattern / OO metric | WMC | CBO | CAM | TPC | APC | ATC | VTC | EPM |
|---|---|---|---|---|---|---|---|---|
| $AP_1$ | **0.95** | **0.23** | **-0.80** | **0.75** | -0.006 | **0.18** | **0.24** | **0.34** |
| $AP_2$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $AP_3$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $AP_4$ | **0.61** | 0.12 | **-0.53** | **0.49** | 0.03 | -0.01 | **0.25** | **0.46** |
| $AP_5$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $AP_6$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

(b) Correlation between OO metrics and anti-patterns for Java2WS.

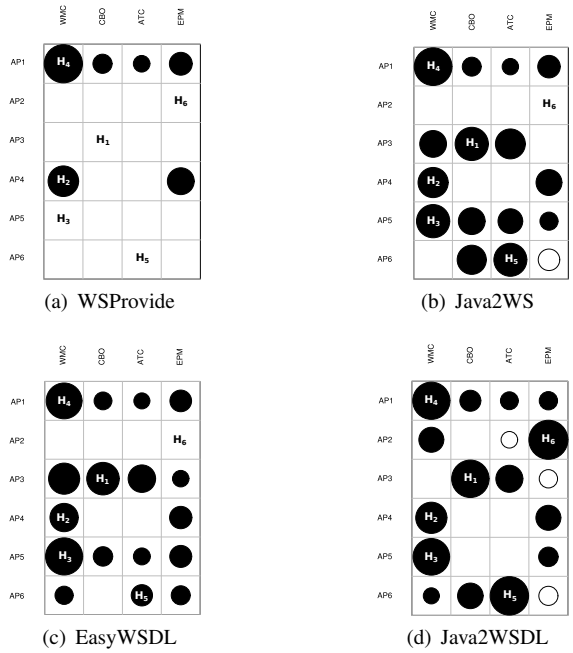
| Anti-pattern / OO metric | WMC | CBO | CAM | TPC | APC | ATC | VTC | EPM |
|---|---|---|---|---|---|---|---|---|
| $AP_1$ | **0.95** | **0.24** | **-0.80** | **0.75** | -0.002 | 0.18 | **0.24** | **0.34** |
| $AP_2$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $AP_3$ | **0.47** | **0.75** | **-0.46** | **0.48** | **0.21** | **0.60** | **-0.13** | -0.11 |
| $AP_4$ | **0.61** | 0.15 | **-0.55** | **0.50** | 0.05 | 0.007 | **0.27** | **0.45** |
| $AP_5$ | **0.74** | **0.48** | **-0.54** | **0.61** | 0.07 | **0.41** | 0.06 | **0.22** |
| $AP_6$ | 0.02 | **0.56** | -0.04 | 0.09 | 0.18 | **0.73** | **-0.26** | **-0.30** |

(c) Correlation between OO metrics and anti-patterns for EasyWSDL.

| Anti-pattern / OO metric | WMC | CBO | CAM | TPC | APC | ATC | VTC | EPM |
|---|---|---|---|---|---|---|---|---|
| $AP_1$ | **0.85** | **0.21** | **-0.72** | **0.69** | -0.01 | **0.17** | **0.24** | **0.31** |
| $AP_2$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $AP_3$ | **0.64** | **0.70** | **-0.59** | **0.67** | **0.31** | **0.50** | 0.13 | **0.18** |
| $AP_4$ | **0.53** | **0.08** | **-0.43** | **0.36** | -0.08 | -0.06 | **0.38** | **0.33** |
| $AP_5$ | **0.91** | **0.25** | **-0.68** | **0.72** | 0.01 | **0.19** | **0.23** | **0.32** |
| $AP_6$ | **0.22** | -0.05 | **-0.21** | **0.13** | -0.08 | **0.30** | **0.42** | **0.23** |

(d) Correlation between OO metrics and anti-patterns for Java2WSDL.

| Anti-pattern / OO metric | 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** |




(a) WSProvide (b) Java2WS

(c) EasyWSDL (d) Java2WSDL

**Figure 3**: Correlation between OO metrics and anti-patterns: Graphical representation.

From Tables 5a, 5b, 5c and 5d, it can be observed that there is a high statistical correlation between a sub-set of the analyzed metrics and the anti-patterns, along with a certain level of consistency throughout the different generation tools. 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*, *Enclosed data model*, *Low cohesive operations in the same port-type* and *Redundant data models*, independently of the tool employed with the exception of WSProvide for the last two anti-patterns for the detection issues already mentioned. Additionally, as expected, ATC and EPM are the best predictors for the two remaining anti-patterns, i.e., *Empty messages* and *Whatever types*.

It is also worth noting that there are two other OO metrics that are highly correlated to several anti-patterns for every generation tool, namely CAM and TPC. However, as shown in Table 4, these two metrics present high correlation factors with WMC and therefore are likely to measure redundant information. Then, only WMC needs to be considered. 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 the experimental data employed.

### *4.2 Code refactorings and discussion*

The correlation among the WMC, CBO, ATC and EPM metrics and the anti-patterns, which is statistically significant for the analyzed Web Service data-set proves that, in

practice, an increment/decrement of the metric values taken on the code of a code-first Web Service directly affects anti-pattern occurrence in its generated WSDL. We then studied some source code refactorings driven by these metrics on the 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 refactored data-sets as $DS_{WMC}$, $DS_{CBO}$, $DS_{ATC}$, $DS_{EPM}$ and $DS_{ALL}$.

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 for the Java primitive type String. In a third 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 original services in order to reduce ATC by, basically, replacing generic arguments with concrete ones. 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 data-set including all the above code modifications. Tables 6a, 6b, 6c and 6d show the impact of the refactoring process on the anti-patterns for each generation tool. Note that only those anti-patterns whose occurrences could be analyzed by the anti-pattern detection tool (discussed in Section 4.1) are shown. Additionally, since OO metrics are independent of the library used for WSDL generation their impacts are showed separately in Table 7.

From the results presented it can be seen that, for all the generation tools employed, the 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*, *Low cohesive operations in the same port-type* and *Redundant data models* anti-patterns of 47.83%, 72.17% and 71.07%, respectively. Similar results were obtained when refactoring by considering CBO, EPM or ATC metrics producing an average decrement of the *Empty messages*, *Enclosed data model* and *Whatever types* anti-patterns by 100.00%, 45.59% and 96.71%, 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 increase the number of occurrences of other anti-patterns. To clarify this, let us take for example the case of the CBO metric, which produced a decrease on the *Enclosed data model* anti-pattern but also a considerable increase on the *Redundant data models* anti-pattern. Furthermore, the negative impact of this increment outweighs the benefits of the refactoring since the total number of anti-patterns is higher with respect to the original data-set. This

**Table 6** Implications of the refactorings on the anti-patterns for different code-first tools.

(a) Refactoring: impact on anti-patterns for WSProvide.

| Anti-pattern | Original (average) | $DS_{WMC}$ (average) | $DS_{CBO}$ (average) | $DS_{ATC}$ (average) | $DS_{EPM}$ (average) | $DS_{ALL}$ (average) |
|---|---|---|---|---|---|---|
| Ambiguous names | 24.42 | 12.58 | 24.42 | 24.42 | 24.42 | 12.58 |
| Low cohesive operations in the same port-type | 30.06 | 8.49 | 30.06 | 30.06 | 24.68 | 8.82 |
| Total number of anti-patterns | 54.47 | 21.08 | 54.47 | 54.47 | 49.09 | 21.41 |

(b) Refactoring: impact on anti-patterns for Java2WS.

| Anti-patterns | Original (average) | $DS_{WMC}$ (average) | $DS_{CBO}$ (average) | $DS_{ATC}$ (average) | $DS_{EPM}$ (average) | $DS_{ALL}$ (average) |
|---|---|---|---|---|---|---|
| Ambiguous names | 24.38 | 12.58 | 24.38 | 24.38 | 24.38 | 12.58 |
| Enclosed data model | 22.09 | 13.08 | 17.32 | 21.98 | 22.09 | 8.91 |
| Low cohesive operations in the same port-type | 30.72 | 8.49 | 30.72 | 30.72 | 25.64 | 8.82 |
| Redundant data models | 72.49 | 20.74 | 150.49 | 72.83 | 69.58 | 36.52 |
| Whatever types | 0.58 | 0.32 | 0.47 | 0.02 | 0.58 | 0.00 |
| Total number of anti-patterns | 150.26 | 55.21 | 223.38 | 149.92 | 142.28 | 66.83 |

(c) Refactoring: impact on anti-patterns for EasyWSDL.

| Anti-pattern | Original (average) | $DS_{WMC}$ (average) | $DS_{CBO}$ (average) | $DS_{ATC}$ (average) | $DS_{EPM}$ (average) | $DS_{ALL}$ (average) |
|---|---|---|---|---|---|---|
| Ambiguous names | 28.58 | 14.75 | 28.58 | 28.58 | 28.58 | 14.75 |
| Enclosed data model | 20.72 | 11.62 | 17.34 | 20.60 | 20.72 | 8.92 |
| Low cohesive operations in the same port-type | 32.91 | 7.29 | 32.91 | 32.91 | 33.47 | 9.94 |
| Redundant data models | 60.25 | 17.88 | 103.74 | 61.17 | 60.55 | 25.97 |
| Whatever types | 0.85 | 0.49 | 0.51 | 0.06 | 0.85 | 0.00 |
| Total number of anti-patterns | 143.30 | 52.03 | 183.08 | 143.32 | 144.17 | 59.58 |

(d) Refactoring: impact on anti-patterns for Java2WSDL.

| Anti-pattern | Original (average) | $DS_{WMC}$ (average) | $DS_{CBO}$ (average) | $DS_{ATC}$ (average) | $DS_{EPM}$ (average) | $DS_{ALL}$ (average) |
|---|---|---|---|---|---|---|
| Ambiguous names | 20.02 | 10.79 | 20.02 | 20.02 | 20.96 | 11.24 |
| Empty messages | 0.94 | 0.44 | 0.94 | 0.94 | 0.00 | 0.00 |
| Enclosed data model | 3.28 | 2.61 | 0.04 | 3.25 | 3.28 | 0.01 |
| Low cohesive operations in the same port-type | 24.62 | 8.19 | 24.62 | 24.62 | 19.04 | 6.25 |
| Redundant data models | 52.96 | 15.10 | 132.96 | 53.89 | 57.81 | 34.10 |
| Whatever types | 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 |

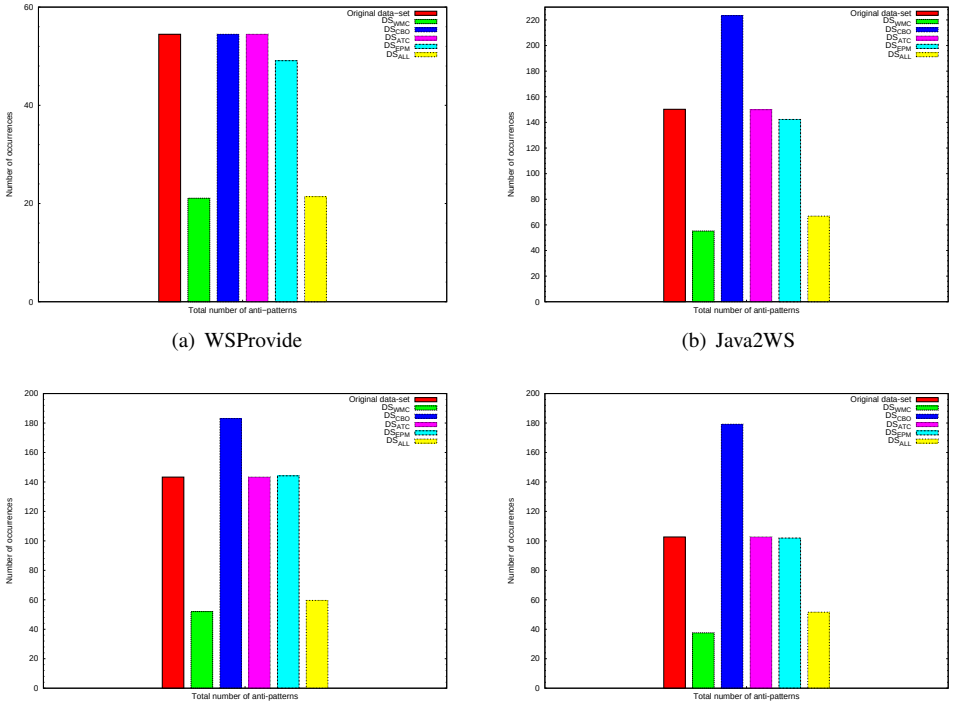
**Table 7** Refactoring: impact on OO metrics.

| Anti-pattern | Original (average) | $DS_{WMC}$ (average) | $DS_{CBO}$ (average) | $DS_{ATC}$ (average) | $DS_{EPM}$ (average) | $DS_{ALL}$ (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 |

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-offs. For example, by decreasing the ATC metric, resulting WSDL documents will present a smaller value for the *Whatever types* anti-pattern than the original WSDL document. However, this will cause an increment of the *Redundant data models* anti-pattern. A similar situation occurs with the EPM metric and the *Empty messages* and *Redundant data models* anti-patterns. 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 7, when the WMC metric is refactored to reduce its value the rest of the OO metrics are indirectly affected and their values are decreased as well. Therefore, all the anti-patterns reduce their total number of occurrences and not just those associated with WMC.

Figure 4 depicts the total average number of anti-patterns of each refactored data-set with respect to the original data-set for each tool. From the experiments it could be observed that paying attention to the WMC metric upon service development allows for services with minimal amount of anti-patterns, thus a simple method for achieving both understandable and discoverable Web Services is given. Moreover, considering CBO renders the associated refactoring counterproductive, while refactoring for ATC or EPM does not have a clear impact on anti-patterns occurrence.

It is worth noting that when all the refactorings were applied on the same data-set ($DS_{ALL}$) the total number of anti-patterns were 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, it can be concluded that if the goal is to minimize the total number of anti-patterns then focusing only on WMC for the refactorings results in principle in the most efficient choice. However, the WMC refactoring splits a service in many smaller services. This leads to the Chatty Services SOA anti-pattern from IBM[6], which basically means realizing a single functional service by implementing a number of Web Services each having a small amount of operations. This in turn results in the implementation of a large number of services leading to degradation in performance because client applications are forced to compose various operations in order to effectively reuse the original (single) service as a whole. In this context, considering a combined and more balanced refactoring strategy similar to $DS_{ALL}$ instead of aggressively refactoring for WMC only might offer WSDL documents with an acceptable number of WSDL anti-patterns while not incurring in the Chatty Services SOA anti-pattern.




(a) WSProvide (b) Java2WS

(c) EasyWSDL (d) Java2WSDL

**Figure 4**: Total number of anti-patterns after refactoring.

## 5 Related work

This work is 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 and Kambhampati (2005), Blake and Nowlan (2008), Pasley (2006), Crasso et al. (2010a), Rodriguez et al. (2010a). In particular, Rodriguez et al. (2010a) subsumes the research listed 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., without anti-patterns). The fact that using improved data-sets allows for better discovery efficiency regardless the approach to Web Service discovery employed, 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 et al. (2010a), when the authors associate anti-patterns with software API design principles. In this sense, it can be said that the approach in this paper 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 this paper, these aspects are quantified in the obtained contracts by means of specific WSDL-level metrics. As discussed earlier, 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, the work in this paper is also related to some efforts that attempt to predict the value of traditional software quality metrics (e.g., number of bugs or popularity) in conventional software based on traditional OO metrics at implementation time Subramanyam and Krishnan (2003), Gyimothy et al. (2005), Meirelles et al. (2010). Particularly, this work extends the approach presented in Mateos et al. (2011) on which a public data-set of around 90 web services is statistically analyzed to determine whether or not a high correlation factor can be established between WSDL anti-patterns and OO metrics. However, in the mentioned approach only the Java2WSDL generation tool was employed for the experimental phase, thus the results presented were not general enough. However, in the mentioned approach only the Java2WSDL generation tool was employed for the experimental phase, thus lacking generality. This paper includes a higher number of Web Services for the data-set and employs several different WSDL generation tools during the experiments, and additionally performs a deeper study of refactorings incidence on anti-patterns manifestation.

## 6 Conclusions

WSDL document specification is one of the most important activities when publishing services in order to get understandable and discoverable (and hence reusable) Web Services Crasso et al. (2010a). It has been shown previously that understandability and discoverability can be improved by avoiding bad WSDL specification practices or anti-patterns Rodriguez et al. (2010a). Mostly, the industry is based on code-first Web Service development, which means that developers first derive a service implementation and then generate the Web Service contracts from the implemented code. In this way, the approach in Mateos et al. (2011) presents a statistical model and preliminary experiments showing that traditional OO metrics can be used as predictors of the level of understandability and discoverability of WSDL documents generated via Java2WSDL.

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. To this end, we based our experiments in four Java-based WSDL generation tools very popular in the industry, and a data-set of 154 real code-first Web Services. We analyzed how certain OO metric-driven refactorings in the code of services impact on the number of occurrences of WSDL anti-patterns. Indeed, having WSDL documents as free as possible from WSDL anti-patterns positively impact on service understandability and discoverability Crasso et al. (2010a).

Precisely, it is known that, when developing contract-first Web Services, removing certain 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 Rodriguez et al. (2010c). In these kind of Web Services, anti-pattern avoidance is manually carried out by developers as Web Service contract design comes before implementation. When developing code-first services, as shown by the findings of this paper, these anti-patterns can be removed or mitigated automatically and indirectly based on source code refactorings. We are investigating the real impact of

the different refactorings –particularly the one reducing WMC– and the extent to which they are applied in the effectiveness of Web Service retrieval apart from anti-pattern occurrences. Effectiveness in this context is being measured by using common metrics from the Information Retrieval area such as Recall, Normalized Recall, R-Precision and Precision-at-n.

This paper has conceived WSDL quality as a synonym of WSDL understandability and discoverability, and thus the anti-pattern catalog can be viewed as a mean to quantify these two aspects from WSDL documents. Alternatively, other authors conceive high quality WSDL documents as those having little complexity Sneed (2010) or exposing high levels of maintainability Basci and Misra (2011). In some recent studies it has been additionally have found that OO metrics and their refactorings can be also used to predict the levels of complexity Coscia et al. (2012) and maintainability Ordiales Coscia et al. (2012a) of code-first Web Services. However, there are some difficult trade-offs situations regarding to what extent refactoring should be applied when trying to balance the values of Web Service quality metrics from different catalogs, i.e., when attempting to build a service that is simultaneously understandable and discoverable, not complex, and maintainable. Therefore, addressing this issue is certainly subject of further research.

## Acknowledgments

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

## References

Bansiya, J. and Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17.

Basci, D. and Misra, S. (2011). Metrics suite for maintainability of extensible markup language Web Services. *IET Software*, 5(3):320–341.

Blake, M. B. and Nowlan, M. (2008). Taming Web Services from the wild. *IEEE Internet Computing*, 12:62–69.

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

Ordiales Coscia, J., Crasso, M., Mateos, C., and Zunino, A. (2012). Estimating Web Service interface complexity and quality through conventional object-oriented metrics. In *XV Ibero-American Conference on Software Engineering (CibSe 2012 - formerly IDEAS)*.

Crasso, M., Rodriguez, J. M., Zunino, A., and Campo, M. (2010a). Revising WSDL documents: Why and how. *IEEE Internet Computing*, 14(5):30–38.

Crasso, M., Zunino, A., and Campo, M. (2008). Easy Web Service discovery: A Query-By-Example approach. *Science of Computer Programming*, 71(2):144–164.

Crasso, M., Zunino, A., and Campo, M. (2010b). Combining document classification and ontology alignment for semantically enriching Web Services. *New Generation Computing*, 28:371–403.

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

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., and Zhang, J. (2004). Similarity search for Web Services. In Nascimento, M. A., Özsu, M. T., Kossmann, D., Miller, R. J., Blakeley, J. A., and Schiefer, K. B., editors, *31th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, pages 372–383. Morgan Kaufmann.

Erickson, J. and 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. and Kambhampati, S. (2005). A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., and Cumby, C. (2010). A search engine for finding highly relevant applications. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 475–484. ACM Press.

Gyimothy, T., Ferenc, R., and 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., and Campo, M. (2010). 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 Special Interest Group on Applied Computing, ACM Press.

Mateos, C., Crasso, M., Zunino, A., and Ordiales Coscia, J. (2011). Detecting WSDL bad practices in code-first Web Services. *International Journal of Web and Grid Services*, 7:357–387.

Meirelles, P., Santos, C., Miranda, J., Kon, F., Terceiro, A., and Chavez, C. (2010). A study of the relationships between source code metrics and attractiveness in free software projects. In *Brazilian Symposium on Software Engineering (SBES '10)*, pages 11–20. IEEE Computer Society.

Ordiales Coscia, J., Crasso, M., Mateos, C., Zunino, A., and Misra, S. (2012a). Predicting web service maintainability via object-oriented metrics: A statistics-based approach. In Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A., Taniar, D., and Apduhan, B., editors, *Computational Science and Its Applications - ICCSA 2012*, volume 7336 of *Lecture Notes in Computer Science*, pages 29–39. Springer Berlin / Heidelberg.

Ordiales Coscia, J., Crasso, M., Mateos, C., and Zunino, A. (2012b). An approach to improve code-first Web Services discoverability at development time. In *2012 Web Technology Track (WT) - ACM Symposium on Applied Computing*, pages 638–643, New York, NY, USA. ACM Press.

Pasley, J. (2006). Avoid XML schema wildcards for Web Service interfaces. *IEEE Internet Computing*, 10:72–79.

Rodriguez, J. M., Crasso, M., Zunino, A., and 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.

Rodriguez, J. M., Crasso, M., Zunino, A., and Campo, M. (2010b). Automatically detecting opportunities for web service descriptions improvement. In Cellary, W. and Estevez, E., editors, *Software Services for e-World*, IFIP Advances in Information and Communication Technology, pages 139–150. Springer.

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

Rusu, L., Rahayu, W., and Taniar, D. (2008). Intelligent dynamic XML documents clustering. In *22nd International Conference on Advanced Information Networking and Applications (AINA 2008)*, pages 449 –456. IEEE Computer Society.

Sneed, H. M. (2010). Measuring Web Service interfaces. In *12th IEEE International Symposium on Web Systems Evolution (WSE), 2010*, pages 111 –115.

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. and 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.

Van Engelen, R. and Gallivan, K. (2002). 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.

W3C Consortium (2007). SOAP version 1.2 part 1: Messaging framework. W3C Recommendation, `http://www.w3.org/TR/soap12-part1` .