Int. J. Web and Grid Services, Vol.

Detecting WSDL bad practices in code-first Web Services

Cristian Mateos*, Marco Crasso and Alejandro Zunino

ISISTAN Research Institute – UNICEN University,
Tandil (B7001BBO), Buenos Aires, Argentina.
Fax: +54 (2293) 43-9682

and

Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
E-mail: cmateos@conicet.gov.ar
E-mail: mcrasso@conicet.gov.ar
E-mail: azunino@conicet.gov.ar
Website: http://www.exa.unicen.edu.ar/~cmateos
Website: http://www.exa.unicen.edu.ar/~mcrasso
Website: http://www.exa.unicen.edu.ar/~azunino
*Corresponding author

José Luis Ordiales Coscia

UNICEN University,
Argentina
E-mail: jlordiales@gmail.com

Abstract: Service-Oriented Computing (SOC) allows developers to structure applications as a set of reusable services. Web Services expose their functionality by using Web Service Description Language (WSDL). We found that there is a high correlation between well-known object-oriented metrics taken in the code implementing services and the occurrences of 'anti-patterns' in their WSDLs. We show that some simple refactorings performed early when developing Web Services can greatly improve the quality of WSDL documents. Then, the contribution of this work is a practical approach to guide practitioners in obtaining better WSDL designs that aligns with the technologies and techniques commonly used in the industry for building services.

Keywords: SOC; service-oriented computing; web services; code-first; WSDL specification; web service discovery; object-oriented metrics; WSDL anti-patterns; early detection.

Reference to this paper should be made as follows: Mateos, C., Crasso, M., Zunino, A. and Ordiales Coscia, J.L. (2011) 'Detecting WSDL bad practices in code-first Web Services', *Int. J. Web and Grid Services*, Vol.

Biographical notes: Cristian Mateos received a PhD in Computer Science from the UNICEN in 2008, and his MSc 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 received a PhD 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 received a PhD in Computer Science from the UNICEN in 2003, and his MSc 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.

José Luis Ordiales Coscia is a Mg. candidate at the UNICEN, working under the supervision of Cristian Mateos and Marco Crasso. His magister thesis is about methods to improve the development of service-oriented systems.

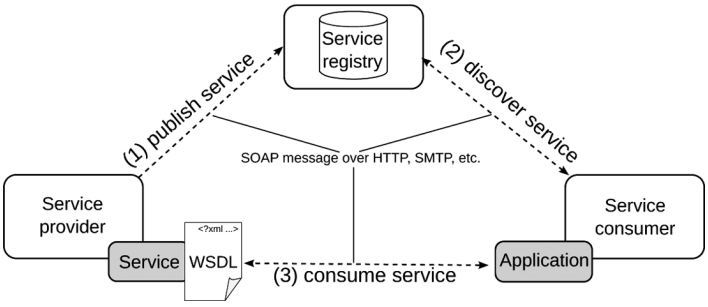
1 Introduction

Service-Oriented Computing (SOC) (Bichler and Lin, 2006; Erickson and Siau, 2008) is a relatively new computing paradigm that has radically changed the way applications are architected, designed and implemented. SOC has mainly evolved from component-based software engineering by introducing a new kind of building block called *service*, which represents functionality that is delivered and remotely consumed using standard protocols. Far from being a buzzword, SOC has been exploited by major players in the software industry and the electronic market including Microsoft, Oracle, Google and Amazon. The applicability of SOC is also breaking through enterprise boundaries and bussiness-to-business interactions since it has become a technological cornerstone of the recent concept of Internet of Things (Lizcano et al., 2009).

From a technological standpoint, the SOC paradigm is commonly materialised through Web Services, i.e., programs with well-defined interfaces that can be published, located and consumed by means of ubiquitous Web protocols (Erickson and Siau, 2008) such as SOAP (W3C Consortium, 2007). The canonical model underpinning Web Services is depicted in Figure 1, and encompasses three basic elements: service providers, service consumers and service registries. A service provider (e.g., a business or an organisation) provides meta-data describing each service, including a technical contract in WSDL (Erl, 2007). WSDL is an XML-based language that allows providers to specify their services' functionality as a set of abstract operations with inputs and outputs, and to associate binding information so that consumers can invoke the offered operations.

To make their WSDL documents publicly available, providers usually employ a specification of service registries called UDDI (OASIS Consortium, 2004), whose

Figure 1 The Web Services model



central purpose is to maintain meta-data about Web Services via a standard relational model. Apart from this model, UDDI defines an enquiry API, in terms of WSDL, for discovering services. Consumers use this API to discover services that match their functional needs, select one, and then consume its operations by interpreting the corresponding WSDL document. Concretely, the enquiry API receives a keyword-based query and in turn returns a list of candidate WSDL documents, which the user who performs the discovery process must analyse. As an alternative to structured Web Service meta-data models like the one featured by UDDI, several syntactic Web Service registries such as Woogle (Dong et al., 2004), WSQBE (Crasso et al., 2008) and seekda! (http://webservices.seekda.com) have emerged. These supports basically work by applying text processing or machine learning techniques, such as XML supervised classification (Crasso et al., 2008) or clustering (Rusu et al., 2008), to build indexes from a collection of WSDL documents. Of course, they also offer keyword-based search of services on top of these indexes (Crasso et al., 2011).

Certainly, service contract design plays one of the most important roles in enabling third-party consumers such as application developers to understand, discover and reuse services (Crasso et al., 2010). On the one hand, unless appropriately specified by providers, service contract meta-data can be counterproductive and obscure the purpose of a service, thus hindering its adoption. Indeed, it has been shown that service consumers, when faced with two or more contracts in WSDL that are similar from a functional perspective, tend to choose the most concisely described (Rodriguez et al., 2010d). A corollary of this is that users will prioritise smaller WSDL documents over larger ones. Moreover, a WSDL description without much comments of its operations can make the associated Web Service difficult to be discovered (Rodriguez et al., 2010d). Particularly, when using UDDI-compliant registries, which are thought to be inspected by human users, service consumers often have to invest much effort into discovering Web Services before finding the functionality they need to outsource. In addition, discovery precision of syntactic registries – which are oriented towards a more automatic search experience compared with UDDI – is harmed when dealing with poorly described WSDL documents (Rodriguez et al., 2010d).

In this sense, as far as we know there has been a unique attempt to integrally study common bad practices or *anti-patterns* found in public WSDL documents,

which instead serves as guidelines, service providers should take into account when specifying service contracts to obtain clear, discoverable services (Rodriguez et al., 2010a). A requirement inherent to applying these guidelines is that services are mostly built in a *contract-first* manner, a method that encourages designers to first derive the WSDL contract of a service and then supply an implementation for it. Then, Rodriguez et al. (2010a) help providers in detecting and removing anti-patterns. However, the most used approach to build Web Services by the industry is *code-first*, which means that one first implements a service and then generates the corresponding service contract by automatically extracting and deriving the interface from the implemented code. This means that WSDL documents are not directly created by humans but are instead automatically derived via language-dependent tools, or by software systems that generate new services at run-time (Sabesan et al., 2010). Consequently, anti-patterns may manifest themselves in the resulting WSDL documents when bad implementation practices are followed (Crasso et al., 2010) or deficient WSDL generation tools are used.

In this paper, we study the feasibility of avoiding WSDL anti-patterns by using object-oriented metrics from the code-implementing services. Basically, the idea is employing these metrics as 'indicators' that warn the user about the potential occurrence of anti-patterns early in the Web Service implementation phase. In this way, this approach would benefit most software practitioners in the industry, which usually rely on code-first service construction. Specifically, through some statistical analysis, we found that there is a statistical significant, high correlation between several traditional and ad-hoc Object-Oriented (OO) metrics and the studied anti-patterns. On the basis of this, we analyse several simple code refactorings that developers can use to avoid anti-patterns in their service contracts. It is worth noting that although our approach is independent of the programming language and the WSDL generation tools used to construct Web Services, we performed our experiments by employing a data set of real Java-based services and Axis' Java2WSDL (http://ws.apache.org/axis/java/user-guide.html#Java2WSDLBuildingWSDLFromJava), which is used by most Java Web Service frameworks to generate WSDL code from Java code.

The rest of the paper is structured as follows. Section 2 gives some background on the anti-patterns present in the above-mentioned guidelines. Then, Section 3 introduces the approach for detecting these anti-patterns at the service implementation phase. Later, Section 4 presents detailed analytical 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 obtained service contracts. Section 5 surveys relevant related works. Lastly, Section 6 concludes the paper.

2 Background

A service development life-cycle, as any other regular kind of software component, consists of several phases. Within these, the service design phase comprises service interface specification using WSDL. Several important concerns, such as granularity, cohesion, discoverability and reusability, should influence

design decisions to result in good service interface designs (Papazoglou and van den Heuvel, 2006). Many of the problems related to the efficiency of standard-compliant approaches to service discovery stem from the fact that the WSDL specification is incorrectly or partially exploited by providers (Rodriguez et al., 2010d).

2.1 The Web Service description language

WSDL is a language that allows providers to describe two parts of a service, namely what it does (its functionality) and how to invoke it. Following the version 1.1 of the WSDL specification, the former part reveals the service interface that is offered to potential consumers. The latter part specifies technological aspects, such as transport protocols and network addresses. Consumers use the functional descriptions to match third-party services against their needs, and the technological details to invoke the selected service. With WSDL, service functionality is described as a set of *port-types*, which arrange different *operations* whose invocation is based on *message* exchange. Messages stand for the inputs or outputs of the operations, indistinctly. Main WSDL elements, such as *port-types*, *operations* and *messages*, must be named with unique names. Optionally, these WSDL elements might contain documentation in the form of comments.

Messages consist of *parts* that transport data between consumers and providers of services, and viceversa. Exchanged data is represented using XML according to specific data-type definitions in XML Schema Definition (XSD) (W3C Consortium, 2009), 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.

2.2 WSDL discoverability anti-patterns

Standard-compliant approaches to Web Service discovery are those based on service descriptions specified in WSDL. 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. Unfortunately, although approaches such as Dong et al. (2004) and Crasso et al. (2008) have been rigorously evaluated and have shown promising results, certainly such results are jeopardised by poorly written WSDL documents. A poorly written WSDL document is one without any proper comments, or containing non-representative or unrelated or redundant keywords. Such a kind of WSDL documents, besides negatively impacting on the retrieval effectiveness of service discovery systems, hinder human discoverers' ability to understand and select the service afterward, as shown in Rodriguez et al. (2010d).

The work published in Rodriguez et al. (2010d) studies recurrent bad practices that take place in a data set of public WSDL documents, measures their impact on both three standard-complaint service registries 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 modelled. Each bad practice description is accompanied by a reproducible solution in Rodriguez et al. (2010d), thus they are called WSDL discoverability anti-patterns, or anti-patterns for short. For the sake of brevity, a sub-set of these anti-patterns is described in Table 1.

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

Anti-pattern	*Occurs when*
Ambiguous names	Ambiguous or meaningless names are used for 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 used for exchanging information are placed in WSDL documents rather than in separate XSD documents
Low cohesive operations in the same port-type	Port-types have weak semantic cohesion
Redundant data models	Many data-types for representing the same objects of the problem domain coexist in a WSDL document
Whatever types	A special data-type is used for representing any object of the problem domain

Source: Adapted from Rodriguez et al. (2010b)

2.3 Approaches to remove WSDL discoverability anti-patterns

Encouraged by the intuitive implications of the inadequate use of WSDL, there are incipient research efforts to provide solutions to anti-patterns. For instance, in Rodriguez et al. (2010a), the authors state that

> "There is no silver bullet to guarantee that potential consumers of a Web Service will effectively discover, understand and access it. However, we have empirically shown that a WSDL document can be improved to simultaneously address these issues by following six steps."

By removing anti-patterns, the proposed guide allows service publishers to improve the cohesion, reusability, readability and discoverability of their service descriptions provided they are able to control their WSDL documents.

Having the control of a WSDL document refers to adhering to a WSDL document construction method known as contract-first. When following this method, providers first create a service interface using WSDL and then implement it using any programming language. Although with this method providers achieve the real importance of WSDL documents as a communication artefact, contract-first is not very popular among developers because the effort it requires is rather bigger than the required by its counterpart, namely code-first. Code-first means implementing a service using any programming language, and then automatically extracting the service interface and translating it into a WSDL document. To understand how this works, let us take the case of Java2WSDL, a software tool that given a Java class produces a WSDL document with operations standing for all public methods declared in the class. Moreover, Java2WSDL associates an XML representation with each input/output method parameter – primitive types or objects – in XSD. One consequence of this WSDL generation method is that any change introduced in service implementations requires the regeneration 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.

To sum up, anti-patterns found in WSDL documents decrease the chance of services to be discovered and reused. The work of Rodriguez et al. (2010d) presents the catalogue of WSDL discoverability anti-patterns, while in Rodriguez et al. (2010a) the authors introduce guidelines to remedy the anti-patterns. Such guidelines are based on refactoring actions for WSDL documents. In this sense, given a WSDL document having anti-patterns, the guidelines encourage providers to refactor the WSDL document (which means to modify it) until all anti-patterns have been removed. Unfortunately, the guidelines can be applied when following contract-first only, but code-first is the de-facto WSDL construction method in the software industry.

As explained in Crasso et al. (2010), the WSDL discoverability anti-patterns are strongly associated with API design qualitative attributes, in the sense that some anti-patterns spring when well-established API design golden rules are broken. For instance, one anti-pattern is associated with tying *port-types* to concrete protocols, which is similar to redefining the interface of a component for each implementation. Another 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 the structured design. In this context, by 'semantically unrelated' we do not refer to operations annotated via unrelated concepts from ontologies, which is the meaning given by researchers in the area of Semantic Web Services (Di Martino, 2009; Kadouche et al., 2009), but to operations whose intented functionality differs. All in all, the main hypothesis of this paper is that it is possible to detect WSDL anti-patterns *early* in the implementation phase by basing on classic API metrics gathered from service implementation and a deep understanding about how WSDL generation tools work. The goal of this work is to detect WSDL discoverability anti-patterns previous to generate WSDL documents, but by basing on service implementations since the code-first method is meant to be supported.

3 The early WSDL bad practices detection approach

The proposed approach aims at allowing providers to prevent their WSDL documents from incurring in the discoverability anti-patterns presented in Rodriguez et al. (2010d) when following the code-first method for building services. To do this, the approach is supported by two facts. First, the approach assumes that a typical code-first tool performs a mapping T , formally:

$$T : C \rightarrow W, \tag{1}$$

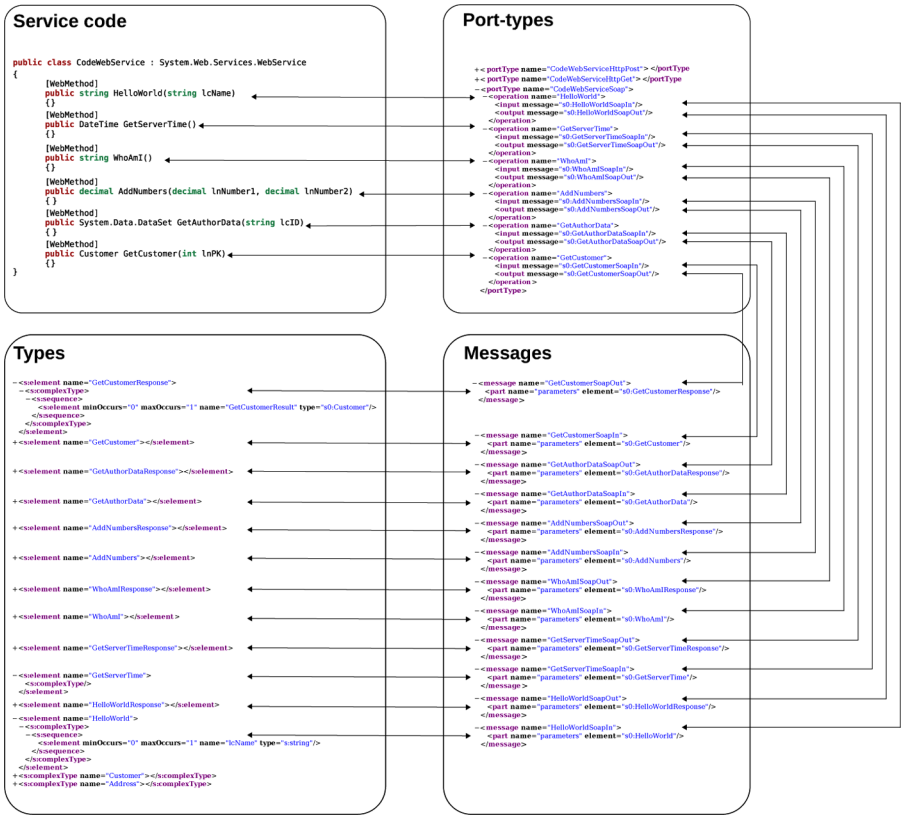
Mapping T from $C = \{M(I_0, R_0), \ldots, M_N(I_N, R_N)\}$ or the frontend class implementing a service to $W = \{O_0(I_0, R_0), \ldots, O_N(I_N, R_N)\}$ or the WSDL document describing the service, generates a WSDL document containing a *port-type* for the service implementation class, having as many *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.

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

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

By basing on the previous two facts, the idea behind the proposed approach is that by employing well-known software engineering metrics on a service code C, a provider might have an estimation of how the resulting WSDL document W

Figure 2 WSDL generation in C# (see online version for colours)



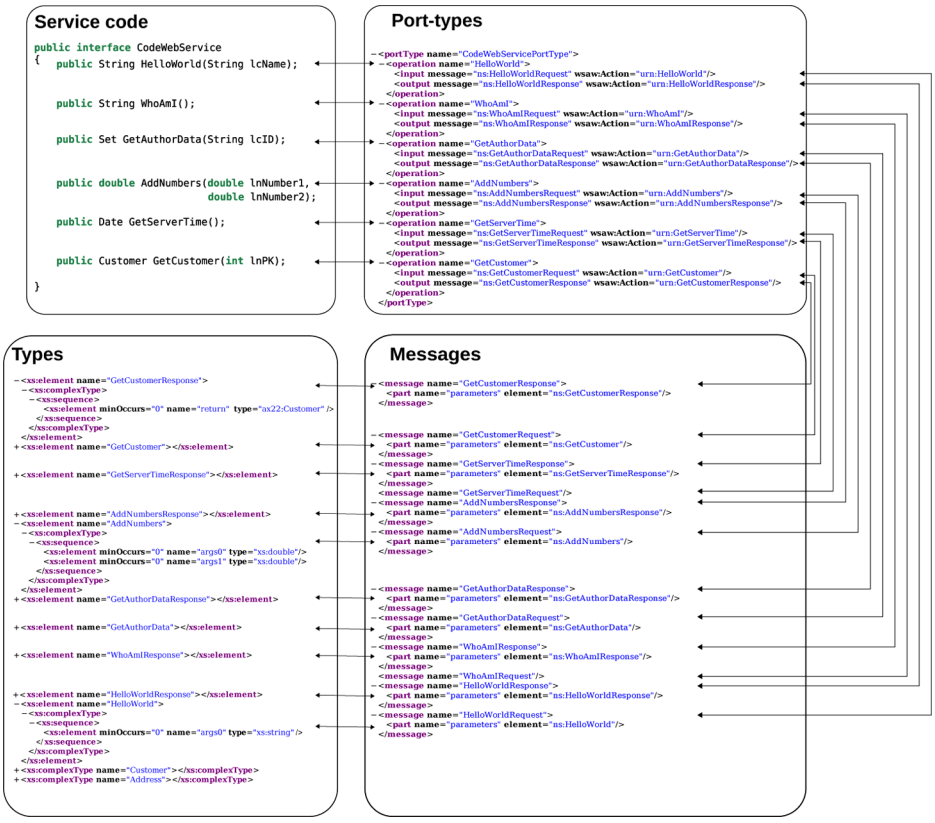
will be like in terms of anti-pattern occurrences, since a known mapping T relates C with W. If indeed such metric/anti-pattern relationships exist, then it would be possible to determine a range of metric values for C so that T generates W without anti-patterns in the best case.

We established several hypotheses by using an exploratory approach to test the statistical correlation among OO metrics and the anti-patterns. As many hypothesis statements arose, here we list those hypotheses that are more relevant to our goals:

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

Basically, Coupling Between Objects (CBO) (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. We believe that increasing the number of external objects that are accessed by service classes may increase the likelihood of data-types definitions within WSDL documents.

Figure 3 WSDL generation in Java (see online version for colours)



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

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

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

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

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

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

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

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

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

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

Hypothesis 7 (H_7)*: The weaker the cohesion of public methods belonging to the class implementing a service* (*LCOM metric*)*, the more frequent the Low cohesive operations in the same port-type anti-pattern occurrences.*

Hypothesis 8 (H_8)*: Same as H_7 but by using a refined version of the LCOM metric known as LCOM3 metric (Henderson-Sellers et al., 1996).*

The rationale behind H_7 and H_8 statements stems from the fact that both the LCOM/LCOM3 metrics and the *Low cohesive operations in the same port-type* anti-pattern deal with the same design attribute, namely cohesion. LCOM (Chidamber and Kemerer, 1994) provides a mean to measure how well the methods of a class are related to each other, and LCOM3 is a refined version, which has been originated because LCOM received some criticism owing to its potential low accuracy under certain circumstances. These metrics deal with cohesion at the class-level, but this anti-pattern deals with cohesion at the WSDL level. Therefore, we believe that poor cohesion may be represented by these metrics and this anti-pattern simultaneously, when the class of the service is non-cohesive under code-first.

Hypothesis 9 (H_9)*: The weaker the cohesion or the stronger the 'tangling' among the methods belonging to the class implementing a service (RFC metric), the*

more frequent the Low cohesive operations in the same port-type anti-pattern occurrences.

Response for Class (RFC) (Chidamber and Kemerer, 1994) counts the methods that can potentially be executed in response to a message received by an object of a given class. We believe that a higher RFC may lead to methods strongly related. On the contrary, we hypothesise that classes with low RFC values will be associated with a higher number of *Low cohesive operations in the same port-type* anti-pattern occurrences.

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

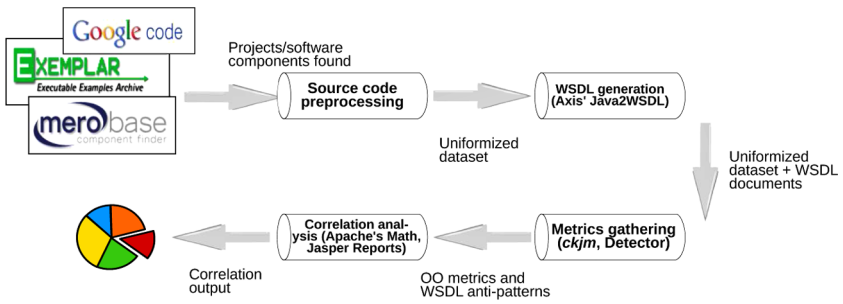
4 Experimental testbed and results

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

The described pipeline has been implemented using software tools for automatising metrics recollection and anti-patterns detection, since the time needed to manually analyse a Web Service project was 2 days/developer and it is an error-prone task. In the former case, we extended *ckjm* (Spinellis, 2005), a Java-based tool that computes a sub-set of the Chidamber-Kemerer metrics (Chidamber and Kemerer, 1994).

For measuring the number of anti-patterns, we employed an automatic WSDL anti-pattern detection tool (Rodriguez et al., 2010c). The WSDL anti-patterns

Figure 4 Software configuration used in the experiments (see online version for colours)



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

In the tests, we used a data set of around 90 different real services whose implementation was collected via two code search engines, namely the Merobase component finder (http://merobase.com) and the Exemplar engine (Grechanik et al., 2010). Merobase allows users to harvest software components from a large variety of sources (e.g., Apache, SourceForge and Java.net) and has the unique feature of supporting interface-driven searches, i.e., searches based on the abstract interface that a component should offer, apart from that of based on the text in its source code. On the other hand, Exemplar relies on a hybrid approach to keyword-based search that combines the benefits of textual processing and intrinsic qualities of code to mine repositories and consequently returns complete projects. Complementarily, we collected projects from Google Code. All in all, the generated data set provided the means to perform a significant evaluation in the sense that the different Web Service implementations came from real-life developers.

Some of the retrieved projects actually implemented Web Services, whereas other projects contained granular software components such as EJBs, which were 'servified' to further enlarge the data set. After collecting the components and projects, we uniformised the associated services by explicitly providing a Java interface to facade their implementations. Each WSDL document was obtained by feeding Axis' Java2WSDL with the corresponding interface. Finally, the correlation analysis was performed by using Apache's Commons Math library (The Apache Software Foundation, 2010), and plots were obtained via JasperReports (Jaspersoft Corporation, 2010).

The rest of the section is structured as follows. Section 4.1 describes the statistical correlation analysis between OO metrics and anti-patterns that were performed on the above-mentioned data set. Lastly, Section 4.2 explores several service refactorings at the source code level and their effect on the bad practices present in the resulting WSDL documents.

4.1 Object-oriented metrics and WSDL anti-patterns: correlation analysis

Broadly, the commonest way of analysing the empirical relation between independent and dependent variables is by defining and statistically testing

experimental hypotheses (Fenton and Pfleeger, 1998). In this sense, we set the six anti-patterns described up to now as the dependent variables, whose values were produced by using the Detector. On the other hand, we used OO metrics as the independent variables, which were computed via the *ckjm* tool.

Furthermore, we employed extra metrics, namely the Lines Of Code (LOC) metric, which counts the number of source code lines in a class (including comments), and two metrics from the work by Bansiya and Davis (2002), i.e., Data Access Metric (DAM) and Cohesion Among Methods of Class (CAM). DAM gives a hint on data encapsulation by computing the ratio of the number of private (protected) attributes to the total number of attributes declared in a class, while CAM computes the relatedness among methods based on the parameter list of these methods. We also included in our study the Morris' Average Method Complexity (AMC) metric (Morris, 1989), i.e., the sum of the cyclomatic complexity of all methods divided by the total number of methods in a class. Finally, as suggested earlier, we extended *ckjm* with a number of ad-hoc measures we thought could be related to the analysed anti-patterns, namely Total Parameter Count (TPC), Average Parameter Count (APC), Abstract Type Count (ATC), Void Type Count (VTC), and Empty Parameters Methods (EPMs).

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

We used Spearman's rank correlation coefficient to establish the existing relations between the two kinds of variables of our model, i.e., the OO metrics (independent variables) and the anti-patterns (dependent variables). Table 3 depicts the correlation factors among the studied OO metrics. The cells values in bold are those coefficients that are statistically significant at the 5% level, i.e., p-value < 0.05, which is a common choice when performing statistical studies (Stigler, 2008). These correlation factors 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 more likely to measure the same underlying dimension (i.e., cohesion, complexity, coupling, etc.). The results presented in Table 3 will be useful later on when we validate the hypothesis presented in Section 3, as we show that some anti-patterns are correlated to several metrics as a result of the statistical dependence between these metrics.

On the other hand, Table 4 shows the correlation between the OO metrics and the anti-patterns, which was obtained by using the same statistical parameters. From the table, it can be observed that there is a high statistical correlation between a sub-set of the analysed metrics and the anti-patterns. Concretely, 2 out of the 14 metrics (i.e., WMC and CBO) are positively correlated to four of the six studied anti-patterns. Furthermore, there are three anti-patterns (*Ambiguous names*, *Enclosed data model* and *Redundant data models*) that are correlated to more than one OO metric. In this sense, to better clarify the analysis of the rationale behind the various high correlation factors, we selected the smallest sub-set of OO metrics that explain the six anti-patterns. This resulted in two sub-sets, namely $< WMC, CBO, ATC, EPM >$ and $< WMC, CAM, ATC, EPM >$. Furthermore, we took the first sub-set as the CBO

Table 2 Descriptive statistics

anti-pattern/Metric	*Minimum*	*Maximum*	*Mean*	*Std. Dev*
Ambiguous names	1.00	247.00	16.31	38.13
Empty messages	0.00	11.00	0.98	1.97
Enclosed data model	0.00	57.00	2.69	7.58
Low cohesive operations in the same port-type	0.00	222.00	7.14	28.61
Redundant data models	0.00	921.00	34.91	134.54
Whatever types	0.00	17.00	0.60	2.00
WMC	1.00	97.00	7.61	14.52
CBO	0.00	31.00	1.88	4.54
RFC	3.00	248.00	34.95	46.76
LCOM	0.00	4753.00	124.78	576.08
LCOM3	0.00	2.00	1.26	0.82
LOC	7.00	4321.00	308.59	694.39
DAM	0.00	1.00	0.38	0.48
CAM	0.13	1.00	0.76	0.25
TPC	0.00	228.00	13.94	31.72
GTC	0.00	15.00	0.31	1.70
EPM	0.00	11.00	1.05	2.11

metric is more popular among developers and is better supported in IDE tools compared with the CAM metric. Moreover, as will be explained in Section 4.2, 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. Sections 4.1.1–4.1.6 present the individual results associated with the validation of the relation between the metrics in the above-mentioned selected subset and the anti-patterns, namely *Ambiguous names* (AP_1), *Empty messages* (AP_2), *Enclosed data model* (AP_3), *Low cohesive operations in the same port-type* (AP_4), *Redundant data models* (AP_5) and *Whatever types* (AP_6).

4.1.1 WMC metric/'Ambiguous names' anti-pattern

Hypothesis H_4 stated that the WMC metric was positively correlated with the *Ambiguous names* anti-pattern. From the correlation analysis shown in Table 4, 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 equal to 0. This shows that hypothesis H_4 is supported by our data, thus confirming its validity.

This result is consistent with the results expected initially. The number of occurrences of the anti-pattern increases when non-representative names are used, both on operation names and on argument names of services. As the value of

Table 3 Correlation between OO metrics

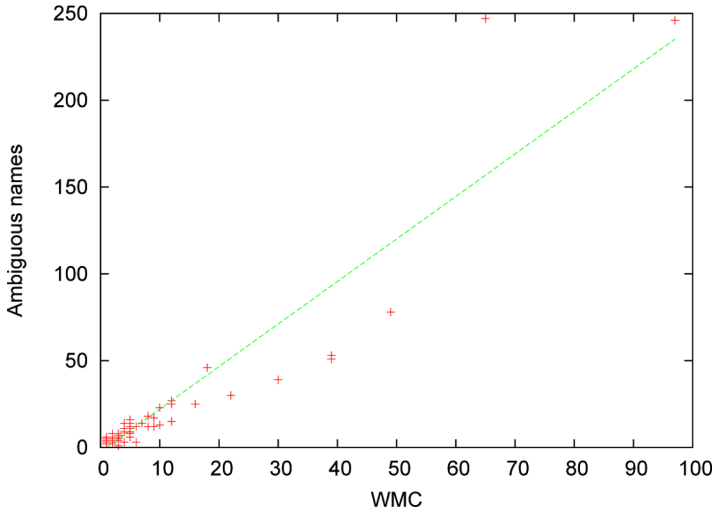
	WMC	*CBO*	*RFC*	*LCOM*	*LCOM3*	*LOC*	*DAM*	*CAM*	*AMC*	*TPC*	*APC*	*ATC*	*VTC*	*EPM*
WMC	1	0.43	**0.52**	0.30	−0.33	**0.50**	0.36	**−0.77**	0.04	**0.81**	0.25	0.27	0.48	**0.54**
CBO	–	1	0.37	0.07	−0.15	0.32	0.15	**−0.65**	0.10	0.33	0.02	0.11	0.34	0.20
RFC	–	–	1	0.25	−0.42	**0.98**	0.44	−0.48	**0.79**	**0.60**	0.41	0.13	0.39	0.21
LCOM	–	–	–	1	0.43	0.22	−0.17	−0.20	0.004	0.36	0.28	0.02	0.14	−0.07
LCOM3	–	–	–	–	1	−0.46	**−0.73**	0.26	**−0.33**	**−0.25**	**−0.08**	**−0.11**	**−0.27**	**−0.43**
LOC	–	–	–	–	–	1	0.48	−0.43	**0.84**	**0.58**	0.40	0.07	0.35	0.21
DAM	–	–	–	–	–	–	1	− 0.25	0.29	0.18	−0.09	0.07	0.19	**0.54**
CAM	–	–	–	–	–	–	–	1	−0.07	**−0.63**	−0.14	−0.24	**−0.55**	−0.49
AMC	–	–	–	–	–	–	–	–	1	0.22	0.30	−0.06	0.08	−0.09
TPC	–	–	–	–	–	–	–	–	–	1	**0.73**	0.20	0.42	0.17
APC	–	–	–	–	–	–	–	–	–	–	1	0.03	0.16	−0.29
ATC	–	–	–	–	–	–	–	–	–	–	–	1	0.37	0.19
VTC	–	–	–	–	–	–	–	–	–	–	–	–	1	0.34
EPM	–	–	–	–	–	–	–	–	–	–	–	–	–	

Table 4 Correlation between OO metrics and anti-patterns

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

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 names. The correlation between the metric and the anti-pattern is depicted in Figure 5.

Figure 5 WMC/'Ambiguous names' correlation (see online version for colours)



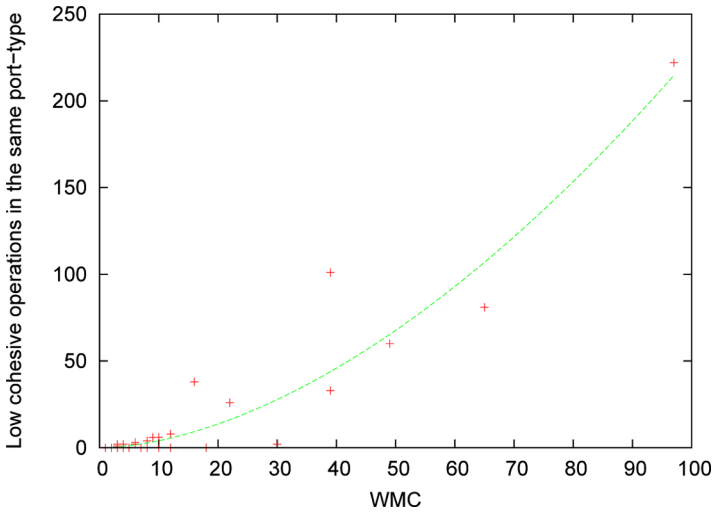
As shown in Table 4, there is also a high correlation factor between the anti-pattern, and the TPC and CAM metrics. This is a direct result of the high correlation factor between these two metrics and WMC. The positive correlation between the WMC and TPC metrics is sound since, in general, as the number of methods of a class increases so does the total number of parameters for that class. On the other hand, by its own definition the CAM metric is inversely proportional to WMC (Bansiya and Davis, 2002), thus causing the negative correlation factor between the two.

4.1.2 WMC metric/'low cohesive operations in the same port-type' anti-pattern

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 4, 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. Figure 6 shows the correlation between the two variables, from which it can be observed that this relation has an exponential nature.

To better justify this exponential tendency, let us consider the following example. Let S_1 be a Web Service with three unrelated methods M_1, M_2 and M_3. In this context, 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 for the two variables would be WMC $= 4$ and *Low cohesive operations in the same port-type* $= 6$.

Figure 6 WMC/'Low cohesive operations in the same port-type' correlation (see online version for colours)



It can be noted that, as we increase the number of methods in the Web Service, the number of occurrences of the anti-pattern tend to increase exponentially with respect to the WMC metric.

4.1.3 WMC metric/'Redundant data models' anti-pattern

Hypothesis H_3 stated that the probability of the *Redundant data models* anti-pattern occurrences increases with the number of public methods, thus implying a positive correlation between the WMC metric and the anti-pattern. From the correlation analysis shown in Table 4, it can be noted that the two variables present a strong positive correlation factor (0.79) and highly significant (p-value $= 0$). This allows us to conclude that the hypothesis is supported by our data. The relation between the metric and the anti-pattern is shown in Figure 7. Moreover, similarly to the relationship between the WMC metric and the *Low cohesive operations in the same port-type* anti-pattern discussed in the previous subsection, the relation between the WMC metric and the *Redundant data models* anti-pattern has an exponential tendency.

This exponentiality arises from the way T-based code-first tools generate the WSDL documents. As we mentioned in Section 3, 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 likelihood of redundant data-type definitions, i.e., 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.

Similarly to the situation discussed in Section 4.1.1 for the *Ambiguous names* anti-pattern, it can be observed from Table 4 that the *Redundant data models* anti-pattern also has a high correlation factor with the TPC metric. This result stems from the high correlation between the WMC and the TPC metric, as depicted in Table 3.

Figure 7 WMC/'Redundant data models' correlation (see online version for colours)

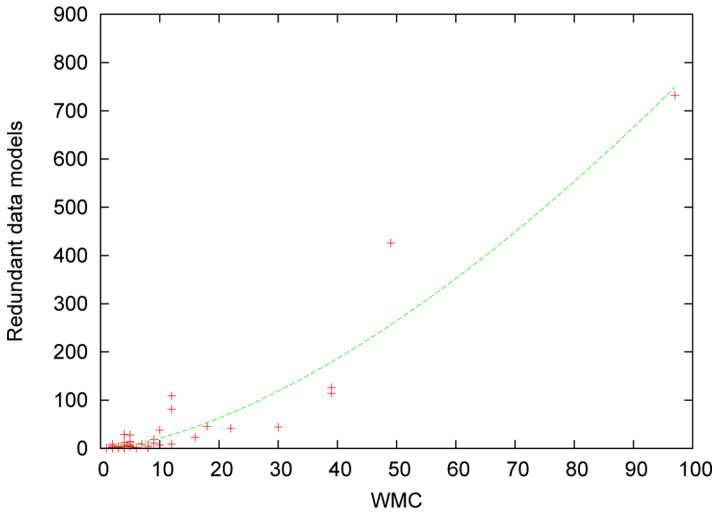
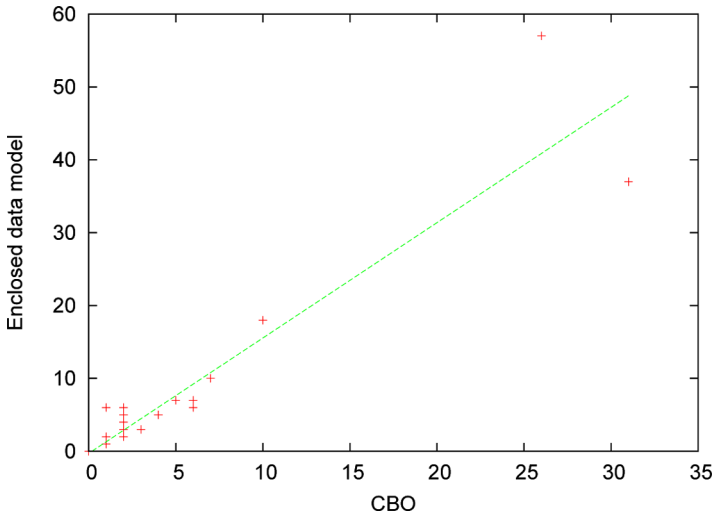


Figure 8 CBO/'Enclosed Data Model' correlation (see online version for colours)



4.1.4 CBO metric/'enclosed data models' anti-pattern

Hypothesis H_1 stated that as the value of the CBO metric increases so does the number of occurrences of the *Enclosed data models* anti-pattern, thus suggesting a positive correlation between the two. From Table 4, it can be seen that there is a statistically significant relationship between the two variables, with a correlation factor of 0.98 and a p-value equal to 0. This shows an almost perfect correlation between the metric and the anti-pattern, i.e., a correlation factor equals to 1. Therefore, we conclude that the hypothesis is supported by our data, thus accepting its validity. Figure 8 depicts the relation between the two variables. Furthermore, it can be seen that this relationship has a linear tendency.

Figure 9 Automatic WSDL generation with 'Whatever types' anti-pattern occurrences (see online version for colours)

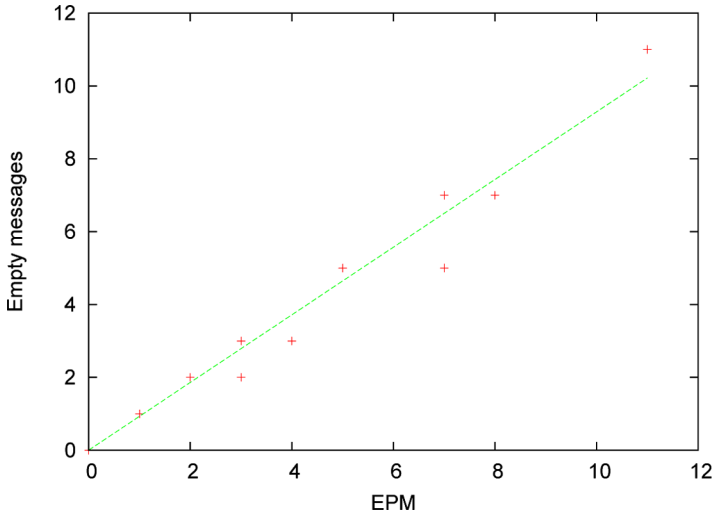
(a) Example service class

```
package org.any;

import java.util.List;
import java.util.HashMap;

public interface SalesManager
{
        public HashMap getSales(List dates, String salesMan);
}
```

(b) Automatically generated WSDL document

```
<wsdl:types>
    <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
                                                targetNamespace="http://any.org">
        <xs:element name="getSales">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="args0" nillable="true" type="xs:anyType"/>
                    <xs:element minOccurs="0" name="args1" nillable="true" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="getSalesResponse">
            <xs:complexType>
                <xs:sequence>
                    <xs:element minOccurs="0" name="return" nillable="true" type="xs:anyType"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:schema>
</wsdl:types>
...
```

Figure 10 EPM *Empty messages correlation* (see online version for colours)



Again, the relation between the metric and the anti-pattern arises 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 lead to a higher number of occurrences of the anti-pattern.

It is worth noting that Table 4 also shows a high correlation factor between the anti-pattern and the CAM metric. Similarly to the situation mentioned in Section 4.1.1 for the WMC, TPC and CAM metrics, this correlation stems from the high correlation factor between the CBO and CAM metrics, as shown in Table 3. This result is due to the fact that both metrics indeed deal with coupling.

4.1.5 ATC metric/'Whatever types' anti-pattern

Hypothesis H_5 stated that an increment in the value of the ATC metric may increase the likelihood of the *Whatever types* anti-pattern occurrences. This suggests a positive correlation between the two. As shown in Table 4, the two variables have a correlation factor of 0.60. Moreover, this correlation is highly significant, with a p-value equal to 0. It can be noted that this correlation factor is the highest for this anti-pattern. Therefore, we conclude that the hypothesis is supported by our data, thus confirming its validity.

The correlation between the metric and the anti-pattern stems from the use of generics and abstract types in the service code. This fact can be better seen on the example shown in Figure 9. Figure 9(a) shows the Java code of a simple service with a single operation that receives a List and a String as input parameters and returns a Hashmap as output parameter. The automatically generated WSDL document using the Java2WSDL tool is shown in Figure 9(b). It can be noted that both the List type and the Hashmap type were mapped onto <anyType> constructors, thus resulting in two occurrences of the *Whatever types* anti-pattern.

4.1.6 EPM metric/'Empty messages' anti-pattern

Hypothesis H_6 stated that a greater number of methods without input parameters increases the probability of the *Empty messages* anti-pattern occurrences, thus suggesting a positive correlation between the EPM metric and the anti-pattern. From the results shown in Table 4, it can be noted that this is clearly the case, as shown by the highly statistically significant relationship between the two variables, with a correlation factor of 0.99 and a p-value equal to 0. These results allow us to conclude that the hypothesis is fully supported by our data. The relation between the metric and the anti-pattern is depicted in Figure 10. It can be observed that the relation presents a strong linear tendency.

The high correlation factor between the two variables is, once again, due to the way T-based code-first tools generate WSDL documents. For those methods that do not receive any parameters, T-based tools still generate an operation element associated with one empty input message element that is not intended to transport any XML data.

4.2 Early code refactorings for improving WSDL documents

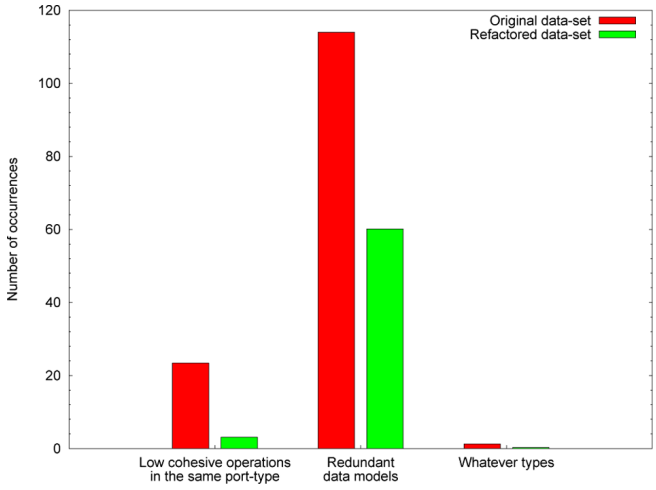
The correlation among the WMC, CBO, ATC and EPM metrics and the anti-patterns, which were found to be statistically significant for the analysed Web Service data set, suggest that, in practice, an increment/decrement of the metric values taken on the code of a Web Service directly affects anti-pattern occurrence in its code-first generated WSDL. Then, we performed some source code refactorings driven by these metrics on our data set so as to quantify the effect on

Table 5 Refactoring: Impact on WMC and its correlated anti-patterns

	Original services (*average*)	*Refactored services* (*average*)
MC	19.32	4.00
Ambiguous names	42.08	42.08
Low cohesive operations in the same port-type	23.40	3.12
Redundant data models	114.00	60.12
Total number of anti-patterns	189.72	135.12

anti-pattern occurrence. For the sake of representativeness, we modified the services that presented all anti-patterns at the same time, which accounted for a 30% of the entire data set. Figure 11 shows the three anti-patterns that were reduced after refactoring.

Figure 11 Refactoring: anti-patterns that were mitigated (see online version for colours)



In a first round of refactoring, we focused on reducing WMC by splitting the services having too much operations into two or more services so that on average the metric in the refactored services represented a 70% of the original value. Table 5 shows the impact on both WMC and its related anti-patterns, i.e., *Ambiguous names*, *Low cohesive operations in the same port-type* and *Redundant data models*. As depicted, on average, these two latter anti-patterns were reduced in 47.26 and 86.66%, respectively. This provides practical evidence to better support part of the correlation analysis of the previous section.

Moreover, from Table 5, it can be seen that the performed refactoring did not affect the average number of occurrences of the *Ambiguous names* anti-pattern. Although in principle this can contradict the analysis made in Section 4.1.1, the true factors that influence the anti-pattern are operation names and argument names of services. The refactoring that was carried out only affected the *number* of operations

of each service, leaving the names of their operations and arguments unmodified. Therefore, the more the WMC, the more the chance of having ambiguous names, however the necessary early refactoring is to modify Web Services so they follow good naming conventions and practices (Rodriguez et al., 2010d,a).

It is worth noting that the refactoring introduced a significant increment of the average number of occurrences of the *Enclosed data model* anti-pattern. Specifically, the original services had on average 6.84 occurrences, against the 20.56 average occurrences in the refactored services. The reason behind this fact is a limitation regarding complex data-type reuse of the current implementation of Java2WSDL, i.e., 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 into let us say 5 new services with 2 operations each, the number of occurrences raises to five since we have 5 services with one occurrence each. In other words, the tool has no sense of such 'data-type globality' 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, however, would produce a less precise and expressive class (and potentially data) model, which attempts against the legibility and clarity of exposed data-types of services. To a certain extent, with this approach, we would trade-off between legibility/clarity and discoverability.

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 data-type globality, but of the Detector. This means that 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 refactoring round, we focused on the ATC metric, which computes the number of parameters in a class that are declared as Object or data structures – i.e., collections – that do not use Java generics. In the latter case, when this practice is followed, these collections cannot be automatically mapped onto concrete XSD data-types for both the container and the contained data-type in the final WSDL. A similar problem arises with parameters whose data-type is Object. In this sense, we modified the services obtained in the previous step to reduce ATC. Note that since ATC and WMC do not conflict between each other and at the same time are correlated to different anti-patterns, results are not affected by the order in which the associated refactorings are performed.

Basically, the applied refactorings was to replace arguments declared as Object with a concrete data-type whenever possible. In addition, although replacing parameters declared as Vector, List, Hashtable, etc., with their generic-aware counterparts, i.e., Vector<X>, List<Y>, Hashtable<K,V> and so on would in theory be another sound refactoring, we decided to replace the former with array structures owing to tool limitations. Overall, by applying these modifications we were able to decrease the number of occurrences of the 'Whatever types'

anti-pattern. Note that the anti-pattern could not be removed completely as the ATC metric only operates at the service interface level. This means that if an interface parameter declared as a concrete data-type X has in turn 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 to 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.

5 Related work

Certainly, our work is to some point related to a number of efforts that can be grouped into two broad classes. On the one hand, there is a substantial amount of research concerning improving services with respect to the quality of the contracts exposed to consumers (Section 5.1). In this sense, we can say that our approach is related to such efforts since we share the same goal, i.e., obtaining more legible, discoverable and clear service contracts.

On the other hand, in our approach, these aspects are quantified in the obtained contracts by means of specific WSDL-level metrics. Furthermore, we found that the values of such metrics can be 'controlled' based on the values of OO metrics taken on the code-implementing services prior to WSDL generation. Then, our approach is also related to some efforts that attempt to predict the value of quality metrics (e.g., number of bugs) in conventional software based on traditional OO metrics at implementation time. These efforts are discussed in Section 5.2.

5.1 Improving WSDL contracts

Several efforts address the problem associated with the quality of WSDL documents from the perspective of discovery. Fan and Kambhampati (2005) surveyed real-world service descriptions and diagnosed how WSDL *documentation* elements are actually employed. Accordingly, the authors conclude that the documentation of 80% out of 640 analysed services has less than 10 words, and as far as 50% of the services have no documentation for any of the offered operations. Regarding WSDL element names, Blake and Nowlan (2008) measured the impact of tendencies for naming on service discovery. To do this, the authors supply a standard-complaint discovery system with heuristics designed for dealing with the identified naming tendencies. As a result, the discovery system under study achieves better retrieval effectiveness than its original version. Another work in this line deals with a common tendency when modelling operation input/output data. Pasley (2006) discussed a common trade-off between extensibility and understandability of data-types defined in XSD. The author explains the impact of using xsd:any and xsd:anyAttribute, which allow developers to leave one or more parts of an XML structure undefined, on the maintainability and discoverability

of Web Services, and suggests that 'any-*' XSD constructors should be avoided.

Rodriguez et al. (2010d) subsumed the research mentioned in the previous paragraph, and also supplied each identified problem with a practical solution, thus conforming a unified catalogue of WSDL discoverability anti-patterns. The importance of WSDL discoverability anti-patterns was initially measured in Rodriguez et al. (2010d) by manually removing anti-patterns from a data set of ca. 400 WSDL documents and comparing the retrieval effectiveness of several syntactic discovery mechanisms when using the original WSDL documents and the improved ones, i.e., the WSDL documents that have been refactored according to each anti-pattern solution. The fact that the results related to the improved data sets surpass those achieved by using the original data set regardless of the approaches to service discovery employed provides empirical evidence that suggests that the improvements are explained by the removal of discoverability anti-patterns rather than the incidence of the underlying discovery mechanism.

Furthermore, the importance of WSDL discoverability anti-patterns has been increasingly emphasised in Crasso et al. (2010), when the authors associate anti-patterns with software API design principles. They state that

> "WSDL documents are not supposed to be big, puzzling, non-cohesive, undocumented, or wrongly named, mainly because their real purpose is to be consumed by other developers. However, ... it seems that the creators of the analysed WSDL documents pass over years of consensus on what is right and wrong when codifying software APIs."

All in all, past research on common bad practices present in WSDL documents, and in particular the anti-pattern catalogue, motivate our work for preventing code-first services from discoverability problems.

5.2 *Using OO metrics as software quality predictors*

OO design metrics, like also Chidamber and Kemerer's suite, have also been employed in Subramanyam and Krishnan (2003) for preventing software defects, specifically those reported by customers when using a software and those identified during customer acceptance testing. The authors state five hypotheses associating one or more metrics with an increase in the number of defects. Here, the dependent variables are the defect count. To test the hypotheses, the authors manually collected metrics from an *e*-commerce suite developed in C++ and Java and compared them against defect resolution logs that were under the control of a configuration management system. Empirical evidence supporting the role of OO design metrics in determining software defects was achieved. Their results, based on metrics data collected on 706 classes in total, including 405 C++ classes and 301 Java classes, indicate that these metrics are significantly associated with defects.

More recently, the correlation between software bugs and Chidamber and Kemerer's suite has been assessed for the well-known Mozilla project in Gyimothy et al. (2005). The authors refer as dependent variables to 8936 different bug entries that had been reported in the bug tracker used during the development of the project. On the other hand, the authors gathered OO metrics, from 3192 C++ classes

as independent variables. For the analysis of relationships between bugs and OO metrics, the authors performed a correlation study. As a result, the authors discuss the importance of each employed metric from an accurate and practical perspective, concluding that CBO seems to be the best in predicting bugs, but LCOM is the most practical since LCOM performed fairly well and it can be easily calculated. All in all, the idea of correlating OO metrics and software defects has proved to be a viable approach to bug detection. For a recent survey including newer efforts in this line, see Catal (2011).

Finally, in Meirelles et al. (2010) the authors evaluated the relations between OO metrics and the popularity or 'attractiveness' of real open source projects. Popularity was quantified based on the number of downloads and members of each project. Furthermore, the authors found that CBO, LCOM and LOC and the total number of code modules were the OO metrics that statistically influenced the independent variables, i.e., downloads and members. Experiments were carried out by using a data set of 6773 projects implemented in C, which were extracted from SourceForge. In the end, the findings were that higher CBO implies more complexity and therefore less popularity, higher LOC suggests more functionality and maturity and hence more attractiveness, and more modules in a project seems to attract more members. This latter fact was due to the independent nature of the code modules of the analysed projects, which arguably allows members to work in parallel on a project's modules without requiring too much cooperation.

Note that, when compared with the approaches analysed at the beginning of this subsection, in which the independent variable has a negative connotation (defects or bugs), Meirelles et al. (2010) included an independent variable whose maximisation is desirable. Likewise, our work also aims at minimising the values of metrics – WSDL anti-pattern occurrences – that measure non-desirable aspects of certain software artefacts, which, therefore, must be minimised.

6 Conclusions and future work

Service contract design, and particularly WSDL document specification, plays one of the most important roles in enabling third-party consumers to understand, discover and reuse Web Services (Crasso et al., 2010). In the previous research, it has been shown that these requirements can be fulfilled provided some common WSDL anti-patterns are not present when deriving WSDL documents or specific corrective actions are carried out when the former situation applies (Rodriguez et al., 2010d). However, an inherent prerequisite for removing such anti-patterns is that services are built in a contract-first manner, by which developers have more control on the WSDL of their services. Mostly, the industry is based on code-first Web Service development, which means that developers first derive a service implementation and then generate the corresponding service contracts from the implemented code.

In this paper, we have focused on the problem of how to obtain WSDL documents that are free from those undesirable anti-patterns when using code-first. On the basis of the approach followed by several existing works in which some quality attributes of the resulting software are predicted during development time, we worked on the hypothesis that anti-pattern occurrence at the WSDL level can

be avoided by basing on the value of OO metrics taken at the code-implementing services. We used well-established statistical methods for coming out with the set of OO metrics that best correlate and explain anti-pattern occurrence by using a data set of real Web Services. To validate these findings from a practical perspective, we also studied the effect of applying simple metric-driven code refactorings to some of the Web Services of the data set on the anti-patterns in the generated WSDLs. Interestingly, we found that these code refactorings, which are very easy to apply by users, effectively reduce anti-patterns, thus improving the resulting service contracts.

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

We are extending this work in several directions. On the one hand, we are studying more refactorings, which in turn could be automated with the help of an IDE. As a starting point, we will use IntelliJ Idea (http://www.jetbrains.com/idea), a Java-based IDE that has many built-in refactoring functions and is designed to be extensible. Second, we will incorporate into our analysis less popular, but nevertheless other WSDL generation tools such as EasyWSDL and JBoss' *wsprovide*. The goal of this task is bringing our findings to a broader audience. Third, we will study the relationships between the anti-patterns and other OO metrics, including traditional metrics such as the ones proposed by Halstead, McCabe, or Henry and Kafura (Tsui and Karam, 2006), and at the same time newer ones (Al Dallal, 2010). This could in turn eventually lead to investigate the effect of other kind of refactorings.

Finally, another research line we are planning to work on relates to service discovery. It is known that, when developing contract-first Web Services, removing WSDL anti-patterns or at least reducing the number of their occurrences increases the retrieval efficiency of syntactic Web Service search engines and thus simplifies discovery (Rodriguez et al., 2010d). In this context, anti-pattern avoidance is manually carried out by developers as they design the contract of their Web Services. In the approach presented in this paper, on the contrary, anti-patterns are removed or mitigated automatically and indirectly based on source code refactorings. In this sense, we will investigate the impact of the different refactorings and the extent to which they are applied in the effectiveness of service retrieval.

Acknowledgement

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

References

Al Dallal, J. (2010) 'Measuring the discriminative power of object-oriented class cohesion metrics', *IEEE Transactions on Software Engineering*, 11 November, 2010, IEEE computer Society Digital Library, IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TSE.2010.97

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

Bichler, M. and Lin, K-J. (2006) 'Service-oriented computing', *Computer* Vol. 39, No. 3, 99–101.

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

Catal, C. (2011) 'Software fault prediction: a literature review and current trends', *Expert Systems with Applications*, Vol 38, No. 4, April, pp.4626–4636.

Chidamber, S. and Kemerer, C. (1994) 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp.476–493.

Crasso, M., Zunino, A. and Campo, M. (2008) 'Easy Web Service discovery: a query-by-example approach', *Science of Computer Programming*, Vol. 71, No. 2, 144–164.

Crasso, M., Rodriguez, J.M., Zunino, A. and Campo, M. (2010) 'Revising WSDL documents: Why and how', *IEEE Internet Computing*, Vol. 14, No. 5, 30–38.

Crasso, M., Zunino, A. and Campo, M. (2011) 'A survey of approaches to Web Service discovery in Service-Oriented Architectures', *Journal of Database Management*, Vol. 22, No. 1, pp.103–134.

Di Martino, B. (2009) 'Semantic Web Services discovery based on structural ontology matching', *International Journal of Web and Grid Services*, Vol. 5, No. 1, pp.46–65.

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. (Eds.): *(e)Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, Morgan Kaufmann, Toronto, Canada, pp.372–383.

Erickson, J. and Siau, K. (2008) 'Web Service, Service-Oriented Computing, and Service-Oriented Architecture: separating hype from reality', *Journal of Database Management*, Vol. 19, No. 3, pp.42–54.

Erl, T. (2007) *SOA Principles of Service Design*, Prentice Hall, Boston, MA, USA.

Fan, J. and Kambhampati, S. (2005) 'A snapshot of public Web Services', *SIGMOD Record*, Vol. 34, No. 1, 24–32.

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

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D. and Cumby, C. (2010) 'A search engine for finding highly relevant applications', *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, Cape Town, South Africa', ACM Press, New York, NY, USA, pp.475–484.

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*, Vol. 31, No. 10, pp.897–910.

Henderson-Sellers, B., Constantine, L.L. and Graham, I.M. (1996), 'Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)', *Object Oriented Systems* Vol. 3, pp.143–158.

Jaspersoft Corporation (2010) jasperreports, http://jasperforge.org/projects/jasperreports

Kadouche, R., Abdulrazak, B., Mokhtari, M., Giroux, S. and Pigot, H. (2009) 'A semantic approach for accessible services delivery in a smart environment', *International Journal of Web and Grid Services*, Vol. 5, No. 2, pp.192–218.

Lizcano, D., Soriano, J., Reyes, M. and Hierro, J.J. (2009), 'A user-centric approach for developing and deploying service front-ends in the future internet of services', *International Journal of Web and Grid Services*, Vol. 5, No. 2, pp.155–191.

Meirelles, Jr., P., C.S., 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', *Brazilian Symposium on Software Engineering (SBES '10)*, Vol. 0, IEEE Computer Society, Los Alamitos, CA, USA, pp.11–20.

Morris, K.L. (1989) *Metrics for Object-Oriented Software Development Environments*, Master's Thesis, M.I.T. Sloan School of Management.

OASIS Consortium (2004) 'UDDI version 3.0.2', *UDDI Spec Technical Committee Draft*, http://uddi.org/pubs/uddi_v3.htm

Papazoglou, M. and van den Heuvel, W-J. (2006) 'Service-oriented design and development methodology', *International Journal of Web Engineering and Technology*, Vol. 2, No. 4, pp.412–442.

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

Rodriguez, J.M., Crasso, M., Mateos, C., Zunino, A. and Campo, M. (2010a) 'The EasySOC project: a rich catalog of best practices for developing web service applications', *XXIX International Conference of the Chilean Computer Science Society (SCCC)*, 15–19 November, pp.33–42.

Rodriguez, J.M., Crasso, M., Zunino, A. and Campo, M. (2010b) '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)* Vol. 9, No. 1, pp.5–23.

Rodriguez, J.M., Crasso, M., Zunino, A. and Campo, M. (2010c) 'Automatically detecting opportunities for Web Service descriptions improvement', in Cellary, W. and Estevez, E. (Eds.): *Software Services for e-World', IFIP Advances in Information and Communication Technology*, Springer, Boston, MA, USA, pp.139–150.

Rodriguez, J.M., Crasso, M., Zunino, A. and Campo, M. (2010d) 'Improving Web Service descriptions for effective service discovery', *Science of Computer Programming*, Vol. 75, No. 11, pp.1001–1021.

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

Sabesan, M., Risch, T. and Luan, F. (2010) 'Automated Web Service query service', *International Journal of Web and Grid Services*, Vol. 6, 400–423.

Spinellis, D. (2005) 'Tool writing: A forgotten art?', *IEEE Software* Vol. 22, pp.9–11.

Stigler, S. (2008) 'Fisher and the 5% level', *Chance*, Vol. 21, pp.12–21.

Subramanyam, R. and Krishnan, M.S. (2003) 'Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects', *IEEE Transactions on Software Engineering*, Vol. 29, No. 4, pp.297–310.

The Apache Software Foundation (2010) *Commons-math: the Apache commons Mathematics Library*, http://commons.apache.org/math

Tsui, F.F. and Karam, O. (2006) *Essentials of Software Engineering*, Prentice Hall, Boston, MA, USA.

Van Engelen, R.A. and Gallivan, K.A. (2002) 'The gsoap toolkit for Web Services and peer-to-peer computing networks', *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02), IEEE Computer Society*, Washington, DC, USA, pp.128–135.

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

W3C Consortium (2009) *XML Schema Definition Language (XSD) 1.1 part 1: Structures*, W3C Working Draft, http://www.w3.org/TR/xmlschema11-1