International Journal of Cooperative Information Systems
Vol. 24, No. 2 (2015) 1550004 (38 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218843015500045

# A Stitch in Time Saves Nine: Early Improving Code-First Web Services Discoverability

Cristian Mateos[*,§], Marco Crasso[†], Alejandro Zunino[*]
and José Luis Ordiales Coscia[‡]

[*]*ISISTAN-CONICET*
*Universidad Nacional del Centro de la Provincia de Buenos Aires*
*Campus Universitario, Paraje Arroyo Seco*
*(B7001BBO) Tandil, Buenos Aires, Argentina*
[†]*IBM Research, Argentina*
[‡]*Klarna, Sweden*
[§]*cristian.mateos@isistan.unicen.edu.ar*

Accepted 14 July 2015
Published 31 July 2015

Web Services represent a number of standard technologies and methodologies that allow developers to build applications under the Service-Oriented Computing paradigm. Within these, the WSDL language is used for representing Web Service interfaces, while code-first remains the de facto standard for building such interfaces. Previous studies with contract-first Web Services have shown that avoiding a specific catalog of bad WSDL specification practices, or anti-patterns, can reward Web Service publishers as service understandability and discoverability are considerably improved. In this paper, we study a number of simple and well-known code service refactorings that early reduce anti-pattern occurrences in WSDL documents. This relationship relies upon a statistical correlation between common OO metrics taken on a service's code and the anti-pattern occurrences in the generated WSDL document. We quantify the effects of the refactorings — which directly modify OO metric values and indirectly alter anti-pattern occurrences — on service discovery. All in all, we show that by applying the studied refactorings, anti-patterns are reduced and Web Service discovery is significantly improved. For the experiments, a dataset of real-world Web Services and an academic service registry have been employed.

*Keywords*: Service-Oriented Computing; Web Services; code-first; Web Services Discovery; object-oriented metrics; WSDL anti-Patterns.

## 1. Introduction

The Service-Oriented Computing (SOC)[14] paradigm represents a distinctive way of architecting, designing and implementing applications. SOC has mainly evolved from component-based software engineering by introducing a new kind of building block called *service*, which represents functionality that is described, discovered and remotely consumed by using standard protocols.[59] The common technological choice for materializing this paradigm is Web Services, which are programs with well-defined interfaces that can be published, discovered and consumed via ubiquitous Web protocols.[14] Among them, the SOAP[6] protocol has been to date the standard for invoking Web Services.

The canonical model that underpins Web Services encompasses three elements: service providers, consumers and registries. A service provider, such as a business or an organization, provides meta-data for each service, including a description of its technical contract in Web Service Description Language (WSDL).[15] With WSDL, the functionality of a service is described as a set of abstract operations (called *port-types*) with inputs and outputs. Operations are defined in the XML Schema Definition (XSD) language. In addition, WSDL allows providers to specify binding information so that consumers can actually consume the offered operations. As surveyed in Ref. 11, there are different approaches to describe the capabilities of Web Services,[8,23,34] such as WSDL-S,[50] which combines WSDL with ontological information in the Ontology Web Language (OWL), or the Web Service Modeling Framework (WSMF).[47] However, the most popular approach used in the industry to describe service interfaces is using merely WSDL.[10,11,37,38,44,55] Moreover, Web Service meta-data is stored in service registries, so that consumers can inquiry registries to find services that match their functional needs, select one, and then invoke its operations by interpreting the corresponding WSDL description. At an upper level of abstraction, different languages have been proposed to organize several services forming services choreographies and/or services workflows, e.g. the Web Service Choreography Interface (WSCI), with the goal of building more complex services. The basic blocks needed to build services choreographies and workflows are the static definitions of individual service interfaces, namely the port-types in their WSDL documents.[33]

For the process of discovering and understanding WSDL documents to be effective, providers must pay special attention to how WSDL documents are specified.[9] For the case of contract-first Web Services, in which WSDL interface specification of services comes before service implementation, by considering a well-established catalog of common WSDL bad practices — i.e. anti-patterns — the understandability, legibility and discoverability of WSDL documents can be significantly improved.[44] These anti-patterns represent bad practices when specifying WSDL documents, and relate to the way port-types, operations and messages are structured and specified. The six anti-patterns considered in this paper are classified into three categories: high-level service interface specification (*Enclosed data model*, *Low cohesive operations in the same port-type* and *Redundant data models* anti-patterns), comments and identifiers (*Ambiguous names* anti-pattern) and service data exchange (*Empty messages* and *Whatever types* anti-patterns).

Unfortunately, the most popular approach to build Web Services in the industry is code-first, by which developers first implement a service and then automatically generate the corresponding WSDL document from the implemented code. To this end, tools such as Axis' Java2WSDL[a] are employed. In this paper, we propose an

[a] Java2WSDL, http://ws.apache.org/axis/java/user-guide.html#Java2WSDLBuildingWSDLFromJava.

approach for avoiding anti-patterns for the case of code-first Web Services. Due to its high popularity, and to limit the scope of our research, we focus on Web Services implemented in Java.

Basically, we set forth the hypothesis that it is possible to avoid WSDL anti-patterns by taking into account Object-Oriented (OO) metrics — in particular the well-known metric suite by Chidamber and Kemerer[7] and two more metrics of our own — when implementing services. The metrics quantify conventional software quality attributes (e.g. cohesion or coupling) in an OO application at several levels (i.e. methods, classes and packages). The idea is employing these metrics as indicators that warn the user about the potential occurrence of anti-patterns early in the Web Service implementation phase. Specifically, through statistical analysis, we found that OO metric values associated with a source code implementing a service are significantly correlated to the number of anti-patterns in the WSDL document generated from this code. Then, we experimentally analyzed the effect of applying simple code refactorings to a data-set of public Web Service implementations, which resulted in a significant reduction of WSDL anti-patterns.

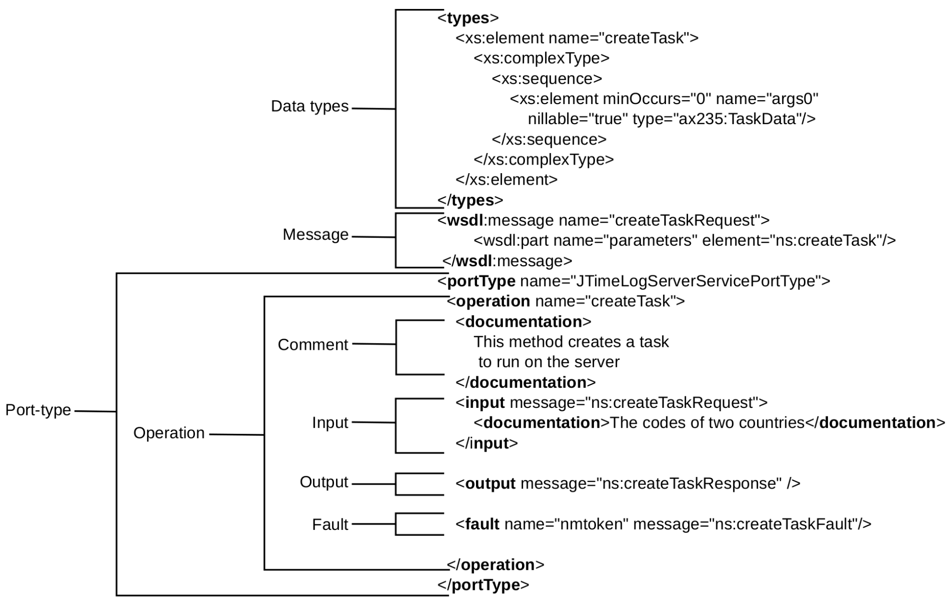
Unlike the study presented in Ref. 44, in which developers have full control over their WSDL documents to *directly* remove any anti-patterns, the approach to anti-pattern remotion in this paper allows providers to *indirectly* avoid some anti-patterns. While in previous work we have preliminary explored the effect of early code refactorings based either on this metric suite[28] and ad-hoc refactorings[45] on the number of anti-patterns in WSDL documents, in this paper we additionally quantify the effect of applying such early code refactorings on service discovery. We conducted further experiments based on the data-set mentioned above and WS-QBE,[10] a Web Service registry based on machine learning and text mining techniques. Experiments showed that WSDL documents derived from code-first Web Services are significantly more discoverable when developed by taking into account our approach. Unlike recent related work of Ref. 29, in which an approach to early anti-pattern remotion based on ad-hoc service code refactorings and a custom WSDL generation tool for the Eclipse IDE was presented, in this paper we aim at avoiding anti-patterns based on popular refactorings that can be used in conjunction with widely-used WSDL generation tools.

The next section overviews the catalog of WSDL anti-patterns and their negative effect on service discovery. Section 3 discusses existing efforts that employ OO metrics as early indicators of potential software quality flaws, and works that attempt to improve the quality of WSDL Web Service interfaces. Section 4 presents our statistical analysis, including the correlation analysis between the studied OO metrics and anti-pattern occurrences, and the effect of some early code refactorings on anti-patterns upon WSDL generation. Section 5 evaluates the effects of these refactorings on Web Service discovery. Section 6 highlights opportunities for future research. Section 7 concludes the paper.

## 2. Background

The service development life-cycle consists of several phases. Within these, the service interface specification phase involves deriving and specifying WSDL documents. Figure 1 presents the general structure of a WSDL document. For contract-first Web Services, developers are strongly involved in WSDL specification and generation, whereas for code-first Web Services WSDL documents are automatically generated from implemented codes via programming language-dependent tools. Regardless of the approach employed to build services, several important concerns, such as granularity, cohesion, discoverability and reusability, should influence decisions upon designing services to result in good service interfaces.[39] Moreover, many of the efficiency problems of standard-compliant approaches to service discovery stem from the fact that WSDL documents are incorrectly specified.[44]

Standard-compliant approaches to Web Service discovery extract keywords from WSDL documents by using classic Information Retrieval techniques, and then model extracted information on inverted indexes or vector spaces.[11] Then, the generated models are used for retrieving relevant service descriptions, i.e. WSDL documents, for a given keyword-based query. Unfortunately, approaches such as Refs. 12 and 10 are jeopardized by poorly written WSDL documents, i.e. those that lack proper comments, contain non-representative, unrelated or redundant keywords, and so on. Besides negatively impacting on the retrieval effectiveness of service registries, some of these issues also hinder human discoverers' ability to make sense of services functionality.[44] In this sense, Rodriguez *et al.*[44] studied recurrent



```
<types>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="args0"
          nillable="true" type="ax235:TaskData"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</types>
<wsdl:message name="createTaskRequest">
    <wsdl:part name="parameters" element="ns:createTask"/>
</wsdl:message>
<portType name="JTimeLogServerServicePortType">
  <operation name="createTask">
   <documentation>
     This method creates a task
      to run on the server
   </documentation>
   <input message="ns:createTaskRequest">
     <documentation>The codes of two countries</documentation>
   </input>

   <output message="ns:createTaskResponse" />

   <fault name="nmtoken" message="ns:createTaskFault"/>

  </operation>
</portType>
```

Fig. 1. Web Services Description Language: Overview and example.

bad practices that take place in public WSDL documents, for example:

- Using meaningless/cryptic or ambiguous terms to name port-types, operations, messages, and message parts. From a semantic perspective, a representative name should precisely describe what its element represents, then meaningless names, such as "in0", "arg1" or "foo", should be avoided. Moreover, if there are two or more elements within a WSDL document standing for the same concept, these elements should be equally named. For instance, if an operation receives user's details as input and another operation produces user's details as output, their corresponding message parts should have the same name. Syntactically, on the other hand, operation names should be in the form: <verb> + <noun>, because an operation is an action. For messages, message parts, or data-types names should be a noun or a noun phrase, otherwise it might mean that a message conveys control information.
- Including *empty* messages in WSDL documents, i.e. input/output messages that do not contain message parts. This happens when there are operations that either receive no input or produce no output, and associated messages are nevertheless specified for the operations.
- Confining (or enclosing) ad-hoc data model definitions in each service description requiring them, and thus data-types can be used only from the operations described in their container WSDL document, preventing the former from being reused from other WSDL documents. Suppose two services for checking stocks when the stock market is open and closed, respectively, with an operation to retrieve market information. One service might define a "StockQuote" data-type while the other service might define a "StockInfo" data-type, however they both represent the same concept but their XSD code is not reused.
- Arranging non-cohesive operations in a single port-type, i.e. operations not belonging to the same domain or not jointly providing a set of semantically related functions. An example is to include operations such as "isAlive", "getVersion" and "ping" in a port-type, though the associated port-type has been designed for providing operations of a particular problem domain, such as offering stock quote information.
- Defining the same data-type more than once in a WSDL document. Suppose a developer combines the enclosed versions of the Web Services for checking stocks at days' open and at days' close into the same WSDL document. Then, two data-types for representing the same object of the problem domain — "StockQuote" and "StockInfo" — will coexist in the WSDL file.
- Using general purpose data-types for representing any object of a problem domain. In general, this typically arises from the use of the XSD constructs `xsd:any` and `xsd:anyAttribute` for representing data-types, which match with any XML structure.

Taking into account these issues, the authors of Ref. 44 measured the impact on service discovery systems and human developers of all the bad practices found,

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

| Anti-Pattern | Occurs When | Affects Discovery, Since... |
|---|---|---|
| Ambiguous names | Ambiguous or meaningless names are used for denoting the main elements of a WSDL document. | Syntactic registries gather and preprocess these names, and instead build ambiguous/poor indexes. |
| Empty messages | Empty messages are used in operations that do not produce outputs nor receive inputs. | Syntactic registries gather and preprocess these names, and instead build ambiguous/poor indexes. |
| Enclosed data model | The data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD documents. | Syntactic registries extract too many terms from data-types associated with messages. Alternatively, data models conceived for being reused may positively impact the precision of registries. |
| Low cohesive operations in the same port-type | Port-types have weak semantic cohesion. | WSDL documents with low-cohesive port-types convey terms that are not always representative of the domain of their associated services. |
| Redundant data models | Many data-types for representing the same objects of the problem domain coexist in a WSDL document. | Redundant data models may produce the same effect as redundant port-types. Besides, big and puzzling WSDL documents are exposed to human discoverers. |
| Whatever types | A special data-type is used for representing any object of the problem domain. | Syntactic registries extract few terms (if any) from Whatever types associated with messages. |

and proposed reproducible solutions to remedy them. The bad practices or anti-patterns listed above, which are the most harmful anti-patterns, are summarized in Table 1. Certainly, these anti-patterns decrease the chance of services to be discovered and reused. Then, these solutions are based on refactoring actions for WSDL documents: given a WSDL document having anti-patterns, the provider can methodically modify it until all anti-patterns have been removed. However, these solutions can be applied when following contract-first only.

However, contract-first is not popular among developers because it requires more effort than code-first: providers must master the WSDL specification and the XSD data-type language. Code-first simply generates WSDLs from existing service code. For example, Java2WSDL can be used to quickly map a given Java class to a WSDL document with operations representing all the public methods. Java2WSDL associates an XML representation with each input/output method parameter — primitive types or objects — in XSD. In the end, providers focus on developing and maintaining service implementations, delegating WSDL/XSD generation to specialized tools during service deployment.

The WSDL discoverability anti-patterns are strongly associated with API design qualitative attributes, since anti-patterns arise when well-established API design golden rules are broken.[9] 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*, even though modules with high cohesion tend to be preferable. Therefore, the main hypothesis of this paper is that it is possible to detect WSDL anti-patterns *early* in the implementation phase by basing on classical OO metrics gathered from service implementation. The goal of this work is to detect anti-patterns before generating WSDL documents, but by basing on service implementations (code-first method). The research hypothesis is that this indirect approach to WSDL anti-pattern remotion has a positive effect on service discovery.

The reader should note at this point that some of the anti-patterns addressed by our approach relate to rules concerning how messages and message parts should be specified and structured within a WSDL document, with the goal of improving service understandability and discoverability. The Web Service Choreography Interface (WSCI)[b] is an effort focusing on providing language constructs to help building service choreographies via message-centered behavioral aspects of individual services such as message ordering, message sequencing, relationships among input and/or output messages, start/end of a given sequence, partial rollback of the execution of a given sequence, and so on. In this way, WSCI describes the dynamic interface of a Web Service participating in a given message exchange by means of reusing the operations defined for the static interface (i.e. port-type), while in this paper we focus on avoiding anti-patterns — which partially involves following some message *formatting* guidelines — to make a Web Service more understandable and discoverable.

### 2.1. *Motivating scenarios*

In many situations, organizations face the need to offer new services to customers in order to stay competitive. Furthermore, some providers, such as large financial organizations and governments, might run legacy systems and at the same time have the necessity of migrating them to newer platforms (i.e. based on Web Services) so other markets and clients can be reached. These two situations represent two common scenarios found in Web Service development in the industry.

In the former scenario, in turn, either new services have to be added to the running system, or portions of such system not exposed as services or "servified" yet are wrapped as Web Services. This is performed by coding the new services from scratch or selecting the modules to servify, and using IDEs to deploy the services in a Web/application container. In Java, for example, IDEs such as Eclipse with its Web Tool Platforms (WTP) plug-in automatically generate most of the necessary

[b]http://www.w3.org/TR/wsci/.

software artifacts, including WSDL documents. As such, these IDEs are designed to conceal WSDL generation from developers.

In the latter scenario, the legacy code (e.g. COBOL) is either completely migrated to modern languages (e.g. Java o C#), which is known as indirect migration,[26] or kept as is but wrapped with a thin software layer (the services) in modern languages, which is referred as direct migration.[26] Due to its simplicity and speed, for migrating legacy systems to Web Services, direct migration is often followed.[27] Again, services are installed with the help of proper IDEs, which automate as much deployment tasks as possible, including WSDL document construction.

In these scenarios, therefore, the task of generating WSDL documents is not controlled by developers, but hidden in tools used to deploy services. These tools, on the other hand, unidirectionally transform source code to WSDL code. Therefore, modifying WSDL documents after services have been deployed leads to inconsistency issues as the source code cannot be automatically updated accordingly. Then, when following code-first service development, the only chance to ensure good WSDL quality is by *preventing* the anti-patterns, instead of *correcting* them as Ref. 44 proposes.

Our research relies on the idea of software quality as the extent to which some specific WSDL-level metrics taken on Web Service contracts present good values. Such metrics basically count anti-pattern occurrences in a WSDL document, thus in this context *good* is indirectly proportional to *low* count of anti-pattern occurrences. Our main research hypothesis is that the values of such metrics can be influenced with the values of OO metrics taken on the code implementing services prior to WSDL generation, which we test using an empirical correlation model. Then, simple refactorings to early alter OO metrics might alter anti-pattern count as well, as a consequence of this correlation, thus improving WSDL understandability and discoverability. Particularly, we focus on two OO metrics from the Chidamber and Kemerer's metric suite, Weighted Methods per Class (WMC) and Coupling Between Objects (CBO), and two metrics from Mateos *et al.*,[28] namely Abstract Type Count (ATC) and Empty Parameters Method (EPM). WMC counts the methods of a class. CBO of a class $C$ measures how many methods/instance variables defined by other classes are accessed from $C$. ATC counts the number of generic/unbounded Java data-types used as parameter of methods. Lastly, EPM counts the number of methods not receiving parameters.

## 3. Related Efforts

Our approach is related to some efforts that try 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 Sec. 3.1. On the other hand, there is substantial research concerning the improvement of services with respect to the quality of the contracts exposed to consumers, and the implication of this in Web Service discovery. These works are discussed in Sec. 3.2. We share the same goal as these efforts, namely obtaining more legible, discoverable and clear service contracts.

### 3.1. *Object-oriented metrics as software quality predictors*

Object-oriented design metrics, and specially the Chidamber and Kemerer's suite,[7] have been extensively used for preventing software defects. For example, in Elish *et al.*[13] the authors state hypotheses associating one or more metrics (i.e. independent variables) with an increase in the number of defects (i.e. bugs reported by customers and other identified during testing). Here, the dependent variables are the number of pre-release faults and post-release faults in software packages. In this context, and from now on, an *independent variable* is a variable that can be altered and manipulated, and a *dependent variable* is the one that is expected to change whenever the independent variable is modified. To test the hypotheses, the authors gathered metrics from Eclipse, one of the largest open source systems, and compared them against bug and fix logs. Besides the Chidamber and Kemerer's suite, the authors also used two package-level metrics, namely Robert Martin's metrics and MOOD.[21]

The correlation between software bugs and the Chidamber and Kemerer's suite has been also assessed for the Mozilla project in Ref. 20. The authors refer as dependent variables to 8936 different bug entries reported in the project bug tracker. The authors gathered OO metrics from 3192 C++ classes as independent variables. For the analysis of relationships between bugs (i.e. dependent variables) 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 Coupling Between Objects (CBO)[7] seems to be the best in predicting bugs, but Lack of Cohesion in Methods (LCOM)[7] 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. A recent survey[42] concludes that 49% of the works that study the correlation of source code metrics and faults base on OO metrics. Within these, the Chidamber and Kemerer's suite is the most frequently used.

Finally, Meirelles *et al.*[31] evaluated the relation 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, LOC, and the total number of code modules where the OO metrics (i.e. independent variables) that statistically influenced the dependent variables, i.e. downloads and members. Experiments were carried out by using a data-set of 6773 projects from SourceForge implemented in C. The findings were that (a) higher CBO implies more complexity and therefore less popularity, (b) higher LOC suggests more functionality and maturity and hence more attractiveness, and (c) more modules in a project seems to attract more members. This latter fact was due to the untangled nature of the code modules of the analyzed projects, which arguably allows members to work in parallel on a project's modules without requiring cooperation.

Note that Ref. 31 includes a dependent variable whose maximization is desirable (attractiveness), whereas in the rest of the approaches mentioned at the beginning of this subsection, the dependent variables have a negative connotation (defects, bugs) and must be minimized. Likewise, our work also aims at minimizing the values of metrics (i.e. WSDL anti-pattern occurrences) that measure non-desirable aspects of certain software artifacts (i.e. WSDL documents).

### 3.2. *Obtaining better WSDL documents*

The growing acceptance of the SOC paradigm motivated research on models and metrics for deriving a comprehensive service-oriented software design methodology along with proper software artifacts.[41] Several efforts have addressed the quality of one of the most important software artifacts for SOC, WSDL service descriptions. The work presented in Ref. 16 surveys real-world service descriptions and diagnoses how WSDL documentation elements are actually employed. Accordingly, the authors conclude that the documentation of 80% out of 640 analyzed services has less than 10 words, and as far as 50% of the services have no documentation for any of the offered operations. Blake and Nowlan[5] measured the impact of naming tendencies within Web Services on service discovery. The authors improved a standard-complaint discovery system with heuristics for dealing with the identified naming tendencies. The improved discovery system achieved better retrieval effectiveness than its original version. Reference 40 in this line discusses 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. The author claims that "any-*" XSD constructors should be avoided.

The work by Rodriguez *et al.*[44] subsumes the research of the previous paragraph by associating each identified problem with a practical solution, thus conforming a unified catalog of WSDL discoverability anti-patterns. The authors manually removed anti-patterns from a data-set of ca. 400 WSDL documents and compared the retrieval effectiveness of several syntactic discovery mechanisms when using the original WSDL documents and the improved ones, i.e. the WSDL documents that had been refactored according to each anti-pattern solutions. The results related to the improved data-sets surpass those achieved by using the original data-set regardless the approaches to service discovery employed. This 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 Ref. 9, 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 analyzed WSDL documents pass over years of consensus on what is right and wrong when codifying software APIs". Past research on common bad practices present in WSDL documents, and in particular the anti-pattern catalog, motivate our work for preventing code-first services from discoverability problems.

In Ref. 28, the authors present a correlation analysis between well-known object-oriented metrics taken in the code implementing services and the occurrences of the anti-patterns of Ref. 44 in their WSDLs documents, showing that some simple refactorings performed early when developing Web Services can reduce the number of anti-patterns occurrences. However, as the author assert, this correlation analysis present an internal threat to validity, since there are more factors that can influence the correlation between service implementation metrics and service interface ones. Therefore, in Ref. 36, the authors have extended the analysis to consider the influence of the tool used for mapping from services implementation onto WSDL documents. In this paper, on the other hand, we improve the studies presented in Refs. 28 and 36 by assessing the implications of refactorings in the discovery of WSDL documents.

Lastly, in Refs. 45 and 29, a tool called AF-JAVA2WSDL to improve WSDL quality for code-first Web Services is presented. The tool implements some ad-hoc refactorings for early removing the anti-patterns in Ref. 44, and a custom Java-to-WSDL mapping tool. Refactorings are driven by automatic suggestions made by AF-JAVA2WSDL on a given service source code, which rely on algorithms based on text mining techniques (e.g. stemming, stop-words removal) and lightweight semantic analysis (e.g. term extraction, synonym/hypernym/hyperonym). These facilities are shipped as a plug-in for the Eclipse IDE. Even when the approach is close to the one presented in this paper, discoverability of the produced services is hardly improved by only relying on the custom mapping tool.[29,45] This forces developers to resort to using such ad-hoc refactorings, which might require coding effort.

The works discussed in this subsection either focus on providing guidelines to improve WSDL documents from a qualitative perspective, or measuring to what extent a WSDL document incurs in the anti-patterns of Ref. 44, which can be partially quantified. Alternatively, there are some incipient works aimed at providing metrics to quantify such aspects, particularly understandability. In this line, Sripairojthikoon and Senivongse[52,53] proposes a readability metric for WSDL documents that exploits the concepts (in ontological terms) in the analyzed service domain knowledge. Given a WSDL document, readability is defined in terms of the use of words regarded as "difficult" and the use of words that are key concepts in the service domain. The authors also outline refactorings to improve readability. In addition Berón *et al.*,[4] presents WSDLUD, a metric for measuring the understanding degree of WSDL descriptions. WSDLUD quantifies the understanding degree values of relevant attributes associated to basic WSDL elements (types, messages, port-types, binding and service) using text-mining techniques and Wordnet,[32] and then aggregates these values into a global score. For example, considered attributes for messages are *Message Documentation Quality*, *Message Name Quality* and *Part Understanding Degree*. WSDL discoverability, which is the scope of this paper, has not been addressed by these metric-oriented approaches to WSDL improvement.

## 4. Correlation Model and Hypotheses

To prevent code-first Web Service descriptions from incurring in the discoverability anti-patterns presented in Ref. 44, our research focuses on addressing these anti-patterns at the service implementation level. We explicitly account for the role of OO metrics and hypothesize that either coupled classes, or weighted classes, or methods returning abstract or void types, are associated with a higher number of anti-pattern occurrences within generated WSDL documents.

We established several hypotheses by using an exploratory approach to test the statistical correlation among OO metrics and the anti-patterns. The hypotheses assume 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. This WSDL document has as many *operations* $O$ as public methods $M$ are defined in class $C$. 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[58] are based on a mapping $T$ for generating WSDL documents from C#, Java and C++, respectively. However, each tool implements $T$ in a particular manner mostly because of the different characteristics of the involved programming languages. For brevity and clarity, in the following subsection we discuss the initial hypotheses that proved to hold after the statistical analysis. The commonest way of analyzing the empirical relation between independent and dependent variables is by defining and statistically testing experimental hypotheses.[17] In this sense, the statistical analysis involved setting the six anti-patterns described up to now as the dependent variables, whose values were produced by using a tool that measures anti-pattern occurrences.[43] On the other hand, we used OO metrics as the independent variables, which were computed via another tool. Finally, we used the Spearman's rank correlation coefficient in order to establish the existing relations between the two kind of variables of our model, i.e. the OO metrics (independent variables) and the anti-patterns (dependent variables).

## 4.1. *Proven hypotheses*

### 4.1.1. *Hypothesis* 1 ($H_1$)

Hypothesis statement: 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.

Our first hypothesis associates the Coupling Between Objects (CBO) metric[7] with the *Enclosed data model* anti-pattern.[44] Basically, CBO counts how many methods or instance variables defined by other classes are accessed by a given class. To show how this metric is computed, consider the service code shown in Fig. 2, which corresponds to one of the services from our data-set. The methods in JTimeLogServer have dependencies with the types UserData (line 3), TaskData (line 4) and StatisticData (line 7). Therefore, the CBO value for the *JTimeLogServer* class is 3. Code-first tools typically map methods parameters in service classes to different XSD definitions in generated WSDL documents. We believe that increasing the number of external objects that are accessed by service classes may increase the number of data-type definitions within WSDL documents.

### 4.1.2. *Hypothesis* 2 ($H_2$)

Hypothesis statement: 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 second hypothesis relates the Weighted Methods Per Class (WMC)[7] metric to the *Low cohesive operations in the same port-type* anti-pattern.[44] WMC counts the methods of a class. For example, the service interface depicted in Fig. 2 defines six public methods (lines 2–7), thus WMC is 6 in this case. We believe that a greater number of methods increases the probability that any pair of them are unrelated, i.e. having weak cohesion. Since code-first tools map each method onto an operation, a higher WMC may lead to WSDL documents having low cohesive operations.

```
1  public interface JTimeLogServer extends java.rmi.Remote {
2    public void setUserId(int in0);
3    public UserData[] getUserList();
4    public TaskData createTask(TaskData in0);
5    public List listProject();
6    public void deleteProject(Object in0);
7    public StatisticData computeStatistics(StatisticData in0);
8  }
```

Fig. 2. Service code example.

### 4.1.3. *Hypothesis* 3 ($H_3$)

Hypothesis statement: 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 third hypothesis states a relationship between the WMC[7] metric and the *Redundant data models*[44] anti-pattern. The number of message elements defined within a WSDL document built under code-first tools, is twice the number of operation elements. 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 these in turn affect the number of operation elements.

### 4.1.4. *Hypothesis* 4 ($H_4$)

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

Similarly to $H_3$, we believe that an increment in the number of methods may lift the number of non-representative names within a WSDL document, since for each method a code-first tool automatically generates in principle five names (one for the operation, two for input/output messages, and two for data-types). Thus, the fourth hypothesis associates the WMC metric with the *Ambiguous names* anti-pattern.

### 4.1.5. *Hypothesis* 5 ($H_5$)

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

Abstract Type Count (ATC) 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. In order to clarify this metric, consider the service code shown in Fig. 2. The listProject method (line 5) returns a List data-type without using generics to declare the type of objects that the list will contain. This means that the list can hold any type of object. Similarly, the deleteProject method (line 6) receives a parameter of type Object, which means that it is impossible to know the parameter's concrete type at compile-time. Then, for the JTimeLogServer service, the value of the ATC metric is 2. We have defined the ATC metric after noting that some code-first tools map abstract data-types and badly defined generics onto xsd:any constructors, which is the root cause of *the Whatever types* anti-pattern.[40,44] Formally, the ATC metric for an interface is

$$\sum_{i=1}^{Op}\sum_{j=1}^{p_i} \text{isAbstract}(i,j),$$

where $Op$ is the number of operations, $p_i$ is the number of input/output parameters of the $i$th operation, and the function $\text{isAbstract}(i, j)$ returns 1 when the type of the $j$th parameter of the $i$th operation is associated with a non-concrete class. Otherwise, the mentioned function returns 0. Thus, the range of the function is $[0, P]$, where $P$ is the total number of parameters.

### 4.1.6. *Hypothesis* 6 ($H_6$)

Hypothesis statement: 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.

The Empty Parameters Methods (EPM) metric counts the number of methods in a class that do not receive parameters. On the example depicted in Fig. 2, this metric takes a value of 2 since the methods getUserList (line 3) and listProject (line 5) do not receive any input parameters. We believe that increasing the number of methods without parameters may increase the likelihood of the *Empty messages* anti-pattern occurrences. This is since code-first tools map this kind of methods onto an operation associated with one input message element not conveying XML data. Formally, the EPM metric for an interface is defined as

$$\sum_{i=1}^{Op} \text{emptyInput}(i),$$

where $Op$ is the number of operations, and the function $\text{emptyInputs}(i)$ returns 1 if the $i$th operation receives no input, and 0 otherwise. Thus, the range of the function is $[0, Op]$, where $Op$ is the total number of operations.

## 4.2. *Data-set and metrics/anti-patterns recollection*

The approach for testing the hypotheses stated above 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. This is done by using regression and correlation methods to validate the usefulness of these metrics for anti-pattern prediction. To perform the analysis, we first gathered a data-set that contained, for each service, its implementation code and dependency libraries needed for generating WSDL documents. A detailed per-service report of the statistical correlation between OO metrics taken on the implementation code and anti-pattern occurrences present in the WSDL documents was calculated. Both the software and the data-set used in the experiments are available upon request.

Report calculation has been done by using software tools for obtaining metrics and detecting anti-patterns. In the former case, we extended *ckjm*,[51] a Java-based tool that computes a sub-set of the Chidamber–Kemerer metrics.[7]

For measuring the number of anti-patterns, we employed an automatic WSDL anti-pattern detection tool[43] or Detector for short. The Detector is a software for automatically checking whether a WSDL document has the anti-patterns of Ref. 44 or not. The Detector receives a given WSDL document as input, and uses heuristics for returning a list of anti-pattern occurrences. As these heuristics are based on the different anti-pattern definitions, there are two groups of heuristics, namely Evident and Not immediately apparent. The Evident heuristics deal with those anti-patterns that can be detected by analyzing only the structure of WSDL documents, like *Empty Messages*, *Enclosed data-types*, *Redundant data models*, and *Whatever types* anti-patterns. The Not immediately apparent heuristics deal with detecting *Low cohesive operations in the same port-type* and *Ambiguous names* anti-patterns because they require a semantic analysis of the names and comments present in WSDL documents. Rodriguez *et al.*[43] combine machine learning and natural processing language techniques to detect the anti-patterns of the second group. Furthermore, reported experiments show that the averaged accuracy of the heuristics was 0.958.[43]

In the tests, we used a data-set of approximately 90 different real services whose implementation was collected via two code search engines: the Merobase[c] component finder and the Exemplar engine.[30] Merobase allows users to harvest software components from a large variety of sources (e.g. Apache, SourceForge and Java.net). It has the unique feature of supporting interface-driven searches, i.e. searches based on the abstract interface that a component should offer. On the other hand, Exemplar relies on a hybrid approach to keyword-based search that combines the benefits of textual processing and the structure of source code to mine repositories and consequently returns complete projects. Complementary, we collected projects from Google Code. All in all, the generated data-set provided the means to perform a proper evaluation in the sense that the different Web Service implementations came from real-life developers.

Some of the retrieved projects 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 uniformized 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 Java interface. Finally, the correlation analysis was performed by using Apache's Commons Math library.[56]

## 4.3. *Empirical correlation model*

Table 2 shows the correlation between the OO metrics associated with the hypotheses presented and the anti-patterns. The cell values in bold are those coefficients which are statistically significant at the 5% level. This level is a common choice when performing statistical studies.[54]

[c]Merobase, http://merobase.com.

Table 2. Most significant correlations between OO metrics and anti-patterns.

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

As shown in Table 2, there is a statistically significant relationship between the CBO metric and the number of occurrences of the *Enclosed data models* anti-pattern, with a correlation factor of 0.98 and an error level of 0. This shows an almost perfect correlation between the metric and the anti-pattern, i.e. a correlation of 1. Therefore, we conclude that the hypothesis $H_1$ is supported by our data, thus accepting its validity. Figure 3 depicts the correlation among the variables using points. Generally speaking, each point $(X,Y)$ represents a Web Service having a value of $X$ for a given OO metric (CBO in this case), and $Y$ occurrences of a certain anti-pattern (enclosed data model in this figure). At the same time, the curve shows the regression between the metric and the anti-pattern. The curve fitting method provided by the Gnuplot[d] library was employed to draw the curve. Furthermore, this relationship has a linear tendency.

The relation between the metric and the anti-pattern arises since the employed code-first tool includes in resulting WSDL documents as many XSD definitions as

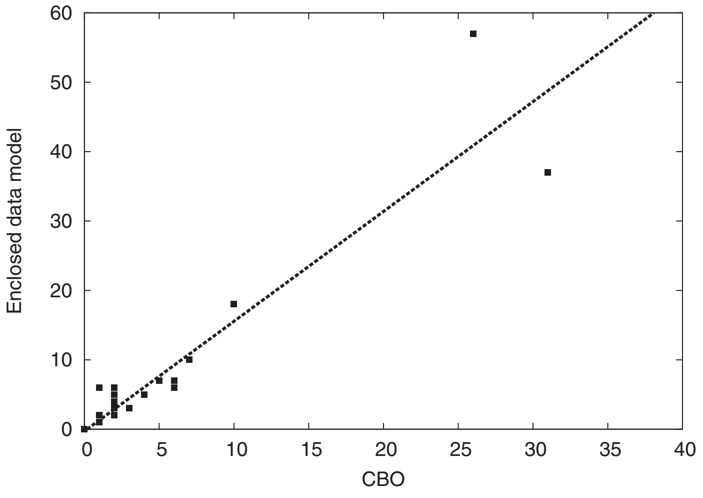


Fig. 3. $H_1$: CBO/enclosed data model correlation.

[d]Gnuplot, http://www.gnuplot.info/.

user defined objects used by the service methods. Then, increasing the value of the CBO metric leads to a higher number of occurrences of the anti-pattern.

The hypothesis $H_2$ states that the likelihood of non-cohesive operations increases with the number of public methods, which suggests a positive correlation between the WMC metric and the *Low cohesive operations in the same port-type* anti-pattern. As shown in Table 2, the correlation factor is the highest for this anti-pattern (0.61) and is also significant (error = 0). This allows us to accept the validity of the hypothesis $H_2$. Figure 4(a) shows the correlation between the two variables, which tends to have 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.

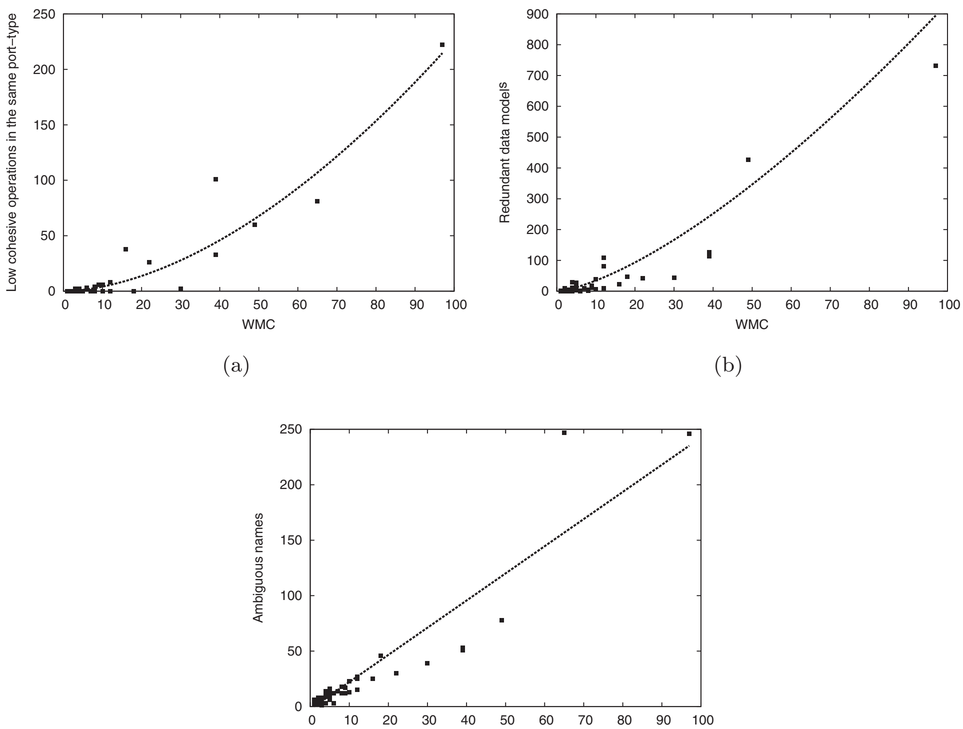


Fig. 4. Correlation between the WMC metric and the anti-patterns. (a) $H_2$: WMC/Low cohesive operations in the same port-type, (b) $H_3$: WMC/Redundant data models and (c) $H_4$: WMC/Ambiguous names.

As we increase the number of methods in the Web Service, the number of occurrences of the anti-pattern tends to increase exponentially with respect to the WMC metric.

The hypothesis $H_3$ states 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 2 follows that the two variables present a positive correlation factor of 0.79 with error $= 0$. Therefore, the hypothesis is supported by our data. Figure 4(b) shows the relation between the metric and the anti-pattern. 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 exponential tendency arises from the way the employed code-first tool generates WSDL documents. This tool defines 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 have the same number and type of input parameters or the same return type, increases exponentially with the number of public methods.

The hypothesis $H_4$ stated that the WMC metric was positively correlated with the *Ambiguous names* anti-pattern. As shown in Table 2, there is a statistically significant relation between the two variables, with a correlation factor of 0.86 and an error of 0. This shows that the 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 argument names of services. As the value of the WMC metric increases, so does the number of operations and arguments, resulting in a higher probability that a sub-set of them use non-representative names. The correlation between the metric and the anti-pattern is depicted in Figure 4(c).

The hypothesis $H_5$ states 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 2, the two variables have a correlation factor of 0.60. Moreover, this correlation is significant, with an error of 0. 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 can be seen in the example shown in Figs. 5 and 6. Figure 5 shows the Java code of a simple service (lines 6–8) with a single operation (line 7) 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 Fig. 6. Note that both the `List`

```
package org.any;

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

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

Fig. 5. Automatic WSDL generation with "Whatever types" anti-pattern occurrences: Example service class.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2    <wsdl:types>
 3      <xs:schema targetNamespace="http://any.org">
 4        <!-- service input data-type -->
 5        <xs:element name="getSales">
 6          <xs:complexType>
 7            <xs:sequence>
 8              <xs:element minOccurs="0" name="args0" type="xs:anyType"/>
 9              <xs:element minOccurs="0" name="args1" type="xs:string"/>
10            <xs:sequence>
11          <xs:complexType>
12        <xs:element>
13        <!-- service output data-type -->
14        <xs:element name="getSalesResponse">
15          <xs:complexType>
16            <xs:sequence>
17              <xs:element minOccurs="0" name="return" type="xs:anyType"/>
18            <xs:sequence>
19          <xs:complexType>
20        <xs:element>
21      <xs:schema>
22    </wsdl:types>
23    ...
```

Fig. 6. Automatic WSDL generation with "Whatever types" anti-pattern occurrences: Automatically generated WSDL document.

type and the Hashmap type were mapped onto <anyType> constructors (lines 8 and 17), thus resulting in two occurrences of the *Whatever types* anti-pattern.

The hypothesis $H_6$ states 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. This is clear in Table 2, as shown by the highly statistically significant relationship between the two variables, with a correlation factor of 0.99 and an error of 0. Therefore, the hypothesis is supported by our data. The relation between the metric and the anti-pattern is depicted in Fig. 7, and presents a strong linear tendency.

The high correlation factor between the two variables is due to the way code-first tools generate WSDL documents. For those methods that do not receive any

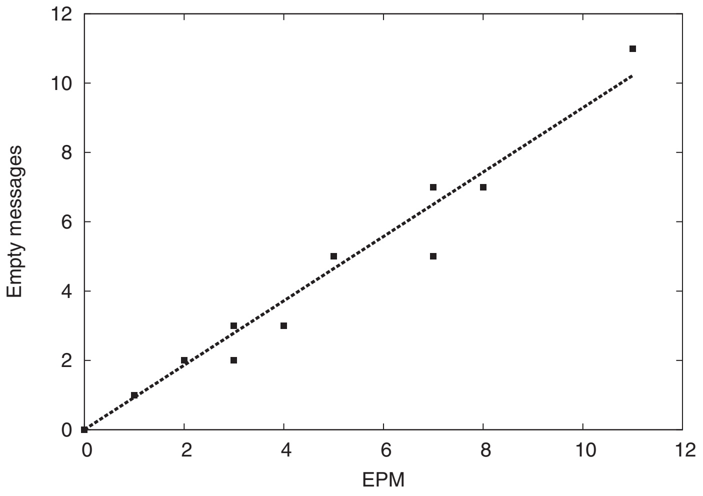


Fig. 7. $H_6$: EPM/Empty messages correlation.

parameters, tools still generate an operation element associated with one empty input message element that is not intended to transport any XML data.

### 4.4. *Early code refactorings for improving WSDL documents*

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 suggests that, in practice, an increment/decrement of the metric values taken on a Web Service code directly affects anti-pattern occurrence in its generated WSDL. Then, we performed some source code refactorings driven by these metrics on our data-set so as to quantify the effect on anti-pattern occurrence. For the sake of representativeness, we modified the services that presented all anti-patterns at the same time, which accounted for a 30% of the entire data-set.

In a first round of refactoring, we focused on reducing WMC. This was done by applying the refactoring associated to WMC in practice: splitting classes having too many operations into two or more classes. In this paper, this means splitting the services having too many operations into two or more services. Since *many* depends on the context in which such refactoring is applied, we decided to split services so that on average the metric in the refactored services represented a 70% of the original value. This resulted in an overall WMC decrement of 79.29% in the dataset. Table 3 shows the impact on the anti-patterns related to WMC, 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 subsection.

Table 3. First round of refactoring: Impact on the anti-patterns correlated to WMC.

| Metric/Anti-Pattern | % decrement |
|---|---|
| Ambiguous names | 0.00 |
| Low cohesive operations in the same port-type | 86.66 |
| Redundant data models | 47.26 |

The modifications introduced a significant increase of the average number of occurrences of the *Enclosed data model* anti-pattern. Specifically, the original services had on average 6.84 occurrences, against 20.56 average occurrences in the refactored services. This stems from 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 if the service is refactored into five new services with two operations each, the number of occurrences of *Enclosed data model* is five since we have five 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 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 understandability and clarity of the exposed data-types of services. To a certain extent, with this approach we would trade-off between understandability/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 two operations both using the defined data-type twice, the Detector counts two anti-pattern occurrences. However, after refactoring, if the service is divided in two 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, 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 needed refactorings are straightforward: 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 be another sound refactoring, we replaced the former with array structures due to WSDL generation tool limitations. Overall, by applying these modifications we decreased the number of occurrences of the "Whatever types" anti-pattern. Note that the anti-pattern could not be removed completely as the ATC metric only operates at the service interface level. This means that if an interface parameter declared as a concrete data-type X has in turn instance variables/generics with non-concrete data-types, the anti-pattern will still manifest.

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 means that the generation tool should not build an empty input message for class methods without parameters. Table 4 shows the three anti-patterns that were reduced after refactoring. It is worth noting that ATC was reduced by 100%, i.e. concrete data-types were used across all refactored service operations.

All in all, for developers to exploit this discussion to reduce anti-patterns in their own services, some general guidelines and considerations for refactoring and mapping service source code to WSDL documents can be derived, which are summarized below:

- The purpose of refactoring is to decrease the value of the discussed OO metrics, since these are positively correlated to anti-patterns, which at the same time should be reduced to increase WSDL quality.
- Depending on the metric being considered, reducing its value might be achieved through different refactorings. Since our goal is to show the effect of reducing metrics on anti-patterns, we consider 1–1 mappings between metrics and possible refactorings for a service $S$: for WMC, we suggest splitting methods in $S$ into several services; for CBO, couplings can be reduced by replacing instance variables in domain objects of $S$ by primitive types; for ATC, we suggest avoiding the use of Object and unbounded types in data structures. Lastly, even when refactorings for EPM exist (e.g. adding default parameters to operations) the associated anti-pattern is introduced by WSDL generation tools.
- The refactorings for WMC, CBO and ATC in the previous point can be applied more or less aggresively: operations in $S$ might be splitted into a different number of services $S_i$; a different number of instance variables of domain objects in $S$

Table 4. Second round of refactoring: Affected anti-patterns.

| Metric/Anti-Pattern | % decrement |
|---|---|
| Low cohesive operations in the same port-type | 13.33 |
| Redundant data models | 52.73 |
| Whatever types | 21.09 |

might be made primitive; different operation parameters of $S$ and instance variables/data structures of domain objects in $S$ might be refactored so they do not contain generic or unbounded data-types.

Since determining the extent to which these refactorings should be applied to result WSDL documents with no anti-pattern occurrences would require a sensitivity analysis, which is out of the scope of this paper, developers should consider other aspects to limit how much refactoring is performed. For WMC, creating many services would lead to fine-grained services, leading to the well-known Chatty Services problem,[e] which negatively affects service consumer application performance since more services need to be called in sequence for accessing same functionality. Moreover, reducing CBO too much could lead to an oversimplified domain object model in the refactored code, which might violate prior design decisions made when building the object model. For ATC, the limit is the effort the developer wants to invest in refactoring, but unlike WMC and CBO, the associated refactoring does not compromise other WSDL quality aspects.

- The proposed refactorings do not conflict between each other, however performing certain refactorings before others might save effort. Particularly, refactoring for CBO/ATC and then considering WMC is desirable, since the former is performed for a single service, which is simpler compared to having multiple services to modify. Moreover, reducing coupling and then removing generic/unbounded data-types is more efficient than the opposite, since refactored data-types might be then replaced by primitive types. Lastly, EPM depends on the WSDL generation tool, and as such should be considered at the end. In conclusion, the suggested refactoring order is CBO, ATC, WMC and EPM.

### 4.4.1. *A practical example*

To conclude this section we will show a practical example of how the discussed refactoring operations can be applied on the service interface depicted in Fig. 2 and how they impact on the resulting WSDL documents. The relevant sections of the original WSDL document generated for this service by the Java2WSDL tool are shown in Fig. 8. There are three defined data-types, namely UserData, StatisticsData and PhaseData (lines 46–48). Therefore, there are three occurrences of the *Enclosed data model* anti-pattern. Similarly, there are two occurrences of the *Whatever types* anti-pattern since the xs:anyType construct is used on the deleteProject and listProjectResponse elements (lines 15 and 41). With respect to the *Redundant data models* anti-pattern, there are three pairs of repeated data-types on the WSDL document: [computeStatistics, computeStatisticsResponse], [createTask, createTaskResponse] and [deleteProject, listProjectResponse]. Finally, while the first five operations of the service (Fig. 2) deal with the manipulation of user data, the last method of this service

[e]http://www.ibm.com/developerworks/webservices/library/ws-antipatterns/.

```
<wsdl:types>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" type="UserData" />
    <xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int" />
    <xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <!-- An occurrence of the 'Whatever␣types' anti-pattern -->
      <xs:element minOccurs="0" name="args0" type="xs:anyType" />
    <xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="StatisticData" />
    <xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" type="StatisticData" />
    <xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="TaskData" />
    <xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" type="TaskData" />
    <xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <!-- Another occurrence of the 'Whatever␣types' anti-pattern -->
      <xs:element minOccurs="0" name="return" type="xs:anyType" />
    <xs:complexType>
  </xs:element>
  <!-- Three occurrences of the 'Enclosed␣data␣model' anti-pattern -->
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...<xs:complexType>
    <xs:complexType name="StatisticData">...<xs:complexType>
    <xs:complexType name="TaskData">...<xs:complexType>
  </xs:schema>
</wsdl:types>
<!-- empty message -->
<wsdl:message name="listProject" />
<wsdl:message name="listProjectResponse" />...</wsdl:message>
<!-- another empty message -->
<wsdl:message name="getUserList" />
<wsdl:message name="getUserListResponse" />...</wsdl:message>
<wsdl:message name="setUserId" />...</wsdl:message>
<wsdl:message name="deleteProject" />...</wsdl:message>
<wsdl:message name="createTask" />...</wsdl:message>
<wsdl:message name="createTaskResponse" />...</wsdl:message>
<!-- Messages corresponding to the unconhesive operation -->
<wsdl:message name="computeStatistics" />...</wsdl:message>
<wsdl:message name="computeStatisticsResponse" />...</wsdl:message>
```

Fig. 8. Original WSDL document generated from Fig. 2.

```
public interface JTimeLogServer_1 extends java.rmi.Remote {
  public void setUserId(int in0);
  public UserData[] getUserList();
  public TaskData createTask(TaskData in0);
}
```

Fig. 9. Resulting service code after the refactorings: First refactored service.

```
public interface JTimeLogServer_2 extends java.rmi.Remote {
  public String[] listProject();
  public void deleteProject(String in0);
  public StatisticData computeStatistics(StatisticData in0);
}
```

Fig. 10. Resulting service code after the refactorings: Second refactored service.

computes statistical information, thus introducing an occurrence of the *Low cohesive operations in the same port-type* anti-pattern.

After applying the proposed refactoring operations, the original service is split into two new services. Figures 9 and 10 show the resulting service codes, while Figs. 11 and 12 depict the refactored WSDL documents. The original service with WMC = 6 (Fig. 2) was divided into two services with WMC = 3 each (Figs. 9

```
<wsdl:types>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int" />
    </xs:complexType>
  </xs:element>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" type="UserData" />
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="TaskData" />
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" type="TaskData" />
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified"
      elementFormDefault="qualified" targetNamespace="http://client.jtl/xsd">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="getUserList" />
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserId">...</wsdl:message>
<wsdl:message name="createTask">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
```

Fig. 11. Resulting WSDL documents after the refactorings: First refactored WSDL document.

```
 1 <wsdl:types>
 2   <xs:element name="computeStatistics">
 3     <xs:complexType>
 4       <xs:sequence minOccurs="0" name="args0" type="StatisticData" />
 5     </xs:complexType>
 6   </xs:element>
 7   <xs:element name="computeStatisticsResponse">
 8     <xs:complexType>
 9       <xs:sequence minOccurs="0" name="return" type="StatisticData" />
10     </xs:complexType>
11   </xs:element>
12   <xs:element name="listProjectResponse">
13     <xs:complexType>
14       <xs:sequence maxOccurs="unbounded" minOccurs="0" name="return" type="xs:string" />
15     </xs:complexType>
16   </xs:element>
17   <xs:element name="deleteProject">
18     <xs:complexType>
19       <xs:sequence minOccurs="0" name="args0" type="xs:string" />
20     </xs:complexType>
21   </xs:element>
22   <xs:schema attributeFormDefault="qualified"
23       elementFormDefault="qualified" targetNamespace="http://client.jtl/xsd">
24     <!-- The six originally defined data-types -->
25     <xs:complexType name="StatisticData">...</xs:complexType>
26     <xs:complexType name="PhaseData">...</xs:complexType>
27     <xs:complexType name="TaskData">...</xs:complexType>
28     <xs:complexType name="ProjectData">...</xs:complexType>
29     <xs:complexType name="RightData">...</xs:complexType>
30     <xs:complexType name="UserData">...</xs:complexType>
31   </xs:schema>
32 </wsdl:types>
33 <wsdl:message name="listProject" />
34 <wsdl:message name="listProjectResponse">...</wsdl:message>
35 <wsdl:message name="deleteProject">...</wsdl:message>
36 <wsdl:message name="computeStatistics">...</wsdl:message>
37 <wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

Fig. 12. Resulting WSDL documents after the refactorings: Second refactored WSDL document.

and 10). Additionally, the ATC metric was reduced to 0 by changing the return type of the listProject method from List to String[] (Fig. 10, line 2) and by changing the parameter of the deleteProject method from an abstract Object type to a concrete String parameter (Fig. 10, line 3). These refactorings resulted in the complete removal of the *Whatever types* anti-pattern, thus the resulting WSDL documents do not have xs:anyType constructs. Similarly, the *Redundant data models* and *Low cohesive operations in the same port-type* anti-patterns were reduced to 2 and 0 occurrences, respectively. On the other hand, the number of occurrences of the *Enclosed data models* anti-pattern was increased by 2, as the first WSDL document defines the data-types UserData (Fig. 11, line 24) and TaskData (Fig. 11, line 25) while the second document defines the original six data-types (Fig. 12, lines 25–30).

## 5. Measuring the Impact of Early Refactoring Anti-Patterns on Discovery

We conducted an experiment to measure the implications on discovery of early detecting anti-patterns in service implementations. This was done by performing the discussed refactorings to remove anti-patterns from target associated WSDL documents, and then discovering these documents. The goal of this experiment was to determine whether placing effort on refactoring service implementations could increase the chance of discovering a service.

Methodologically, the evaluation consisted of three steps. In a first step, code-first WSDL documents were grouped into two groups. One group called "Refactored" contained those WSDL documents generated after applying the refactorings to a sub-set of the service implementations. Another group had the original versions of the improved WSDL documents. We refer to this group as "Original". Second, we supplied a service registry with both groups of WSDL documents. Third, we queried the employed registry using one query per available service operation in the services of the sub-set. For each query we analyzed in which position were retrieved either the original or the refactored WSDL documents, which is formally known as Precision-at-$n$. Precision-at-$n$ computes precision at different cut-off points. For example, if the top five documents are all relevant to a query and the next five are all non-relevant, we have a precision of 100% at a cut-off of five documents but a precision of 50% at a cut-off of 10 documents. Finally, we averaged the results over the total number of queries.

As the reader can observe, the refactored documents were basically WSDL documents whose implementations were modified to take into account not some, but all the refactorings discussed in the previous Section. The reason behind this decision was that, even when some anti-patterns affect service discovery more than others, they all negatively impact on the discoverability of WSDL documents. Hence, the best results in terms of retrieval effectiveness are obtained when removing all the anti-patterns.[44] This means that all the refactorings associated to their correlated OO metrics have to be considered. Therefore, these facts ensure the significance of the results. On the other hand, in practice, modifying service implementations by taking into account all the refactorings is not an expensive task since most of them can be easily performed with the help of modern IDEs.

For the experimentation, a publicly available registry implementation of the approach to service discovery presented in Ref. 10 was employed. The registry can be downloaded from http://sites.google.com/site/easysoc/home/service-registry. As explained in Ref. 10, this registry exploits relevant information contained in WSDL documents. Then, such information is preprocessed using a combination of text mining and machine learning techniques to remove redundant plus non-relevant data and build a vectorial representation of each service, respectively. This is a classical model borrowed from the Information Retrieval area known as Vector Space Model.[48] With this model, documents are seen as collection of terms, whereas each dimension of the space corresponds to a separate term (usually single words). In consequence, documents having similar contents are represented as vectors located near in the space, thus searching related documents translates into searching nearest neighbors in the space. Discoverers can use any form of textual based queries, ranging from single keywords to textual descriptions of their needs, to query the registry. During the discovery process, the registry maps a query onto a vector in the vector space model, then it returns to the discoverer those services whose vectors are near to the query vector.[10] This registry returns an ordered list of candidate WSDL documents, sorted according to how similar to the query are the associated services.

For the sake of fairness we built the employed queries from the source code of original service implementations. We assumed that if developers want to replace an operation with a functional equivalent operation provided by an external service, they will probably use the name of the replaceable operation as a query. This is analogous to the Query-By-Example concept presented in Ref. 10. For example, the query for looking for operations functionally equivalent to an operation whose signature is: "getActiveWorkflows(userID:string)" may be "get active workflows". In fact, the employed registry splits combined words within queries. Following this assumption, 463 queries were built, one per offered operation. Finally, we associated two WSDL documents with each query, one document belonging to the Original group, while another from the Refactored one. For the association we arbitrarily selected the WSDL documents containing the operation needed.

The Precision-at-$n$ results have been calculated for each query with $n$ in $[1, 10]$. We have chosen this window to have a good balance between the number of candidates and the number of relevant candidates retrieved. Moreover, we believe that a developer can easily examine 10 Web Service descriptions. Therefore, by setting $n = 10$, we refer to the actual number of relevant services up to only 10 candidates in the result list. Besides Precision-at-$n$, the mean average precision (MAP), recall and discounted cumulative gain (DCG) measures have been calculated. The MAP measure provides a measure of quality across recall levels. Formally, MAP is defined as

$$\mathrm{MAP} = \frac{\sum_{q=1}^{Q} \mathrm{precision}(q)}{Q},$$

where $Q$ is the number of queries, i.e. 463, and precision$(q)$ is the Precision-at-1 for the query $q$. Then, the MAP metric for the Original data-set was 2%, whereas the MAP metric for the Refactored data-set was 66%. This result provided a global perspective of the performance obtained for each data-set, which allows affirming that refactored WSDL documents surpassed the original ones. On the other hand, Recall was calculated for the window of 10 candidates, which is normally called Recall-at-10. Formally, Recall is defined as

$$\mathrm{Recall} = \frac{Relevant}{Retrieved}.$$

In particular, in our experiments the numerator (*Relevant*) of the above formula could be 0 or 1, i.e. when the target WSDL document is included within the results, and the denominator (*Retrieved*) is always 10. From our experiments, we observed that the original WSDL documents were included in the results list for the 81% of the queries, i.e. Recall $= 0.81$. At the same time, the refactored WSDL documents achieved a Recall of 96%.

The DCG is a measure for ranking quality and measures the usefulness or gain of an item based on its relevance and position in the provided list. A higher DCG value, means that a query returned a list of better ranked candidates. Formally, DCG is defined as

$$\text{DCG} = \text{rel}_1 + \sum_{i=2}^{p} \frac{\text{rel}_i}{\log_2 i},$$

where $p$ is the size of the candidate list, which for these experiments is 10, and $\text{rel}_i$ indicates if the candidate retrieved in the $i$th position of the list was relevant. The DCG values for all queries can be averaged to obtain a measure of the average performance of a ranking algorithm, named normalized DCG (nDCG). We calculated the DCG for each query and then we averaged the values for the 463 queries. Accordingly, the nDCG was 59% and 90%, for the Original and Refactored data-sets, respectively.

Figure 13 depicts the averaged precision-at-$n$ results for the 463 queries by smoothing these results using Bézier curves. Results show that refactored WSDL documents were ranked before their original counterparts. Having a higher precision-at-1 means that a relevant service was retrieved at the top of the result list. Precision-at-1 was 66.1% and 2.6% for refactored and original groups, respectively. In other words, the WSDL documents associated with services whose implementations had been refactored, were ranked first in the 66.1% of the cases. As shown in Fig. 13, 93% of relevant refactored WSDL documents were retrieved at the forth position, in the worst case, whereas a fraction of 64% of relevant Original WSDL documents were retrieved at the same position. Accordingly, discoverers would have to analyze up to only four candidates until finding a relevant service when employing refactored WSDL documents in 93% of the cases.

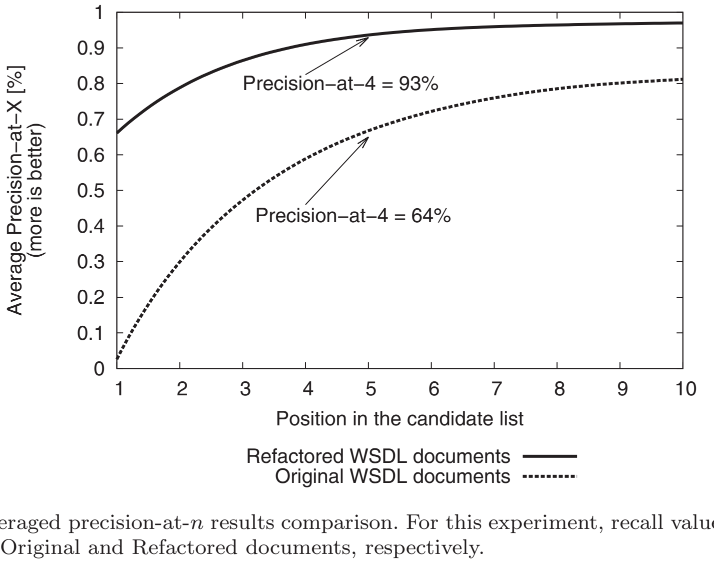


Fig. 13. Averaged precision-at-$n$ results comparison. For this experiment, recall values were 81% and 96% for Original and Refactored documents, respectively.

Clearly, refactored WSDL documents were better ranked compared to Original ones. As supported by different experiments, these results have a great impact on discoverability because users tend to select higher ranked search results.[1] For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the next one is, at most, 60%.[1]

The fact that only Original and Refactored versions of the same WSDL documents coexist in a registry, although useful for comparison purposes, is unrealistic. We have assessed the implications of early detecting anti-patterns and applying the proposed refactorings in a more realistic scenario, by reproducing the same experiment with a data set of ca. 400 publicly available WSDL documents.[22] This data set was published in the employed registry along with the Original and Refactored groups of WSDL documents. Therefore, methodologically, the second experiment was equal to the former except for the second step, which has been modified to simulate a real world scenario. We also calculated the global measures for this experiment. Recall-at-10 values were 76% and 95% for the Original and Refactored data-sets, respectively. The MAP measure values were 2% and 60% for the Original and Refactored data-sets, respectively. The nDCG measure values were 51% and 87%, for the Original and Refactored data-sets, respectively. Again, the global metrics showed a favorable tendency to the Refactored WSDL documents.

Figure 14 shows the Precision-at-$n$ results for the second experiment. Again, the 463 Precision-at-$n$ results have been averaged and smoothed using Bézier curves. The tendency of first ranking Refactored WSDL documents is maintained, though Precision-at-$n$ results fell by 2.85% and 7.02% for the Refactored and Original groups, respectively. This indicates that the discoverability of Original WSDL

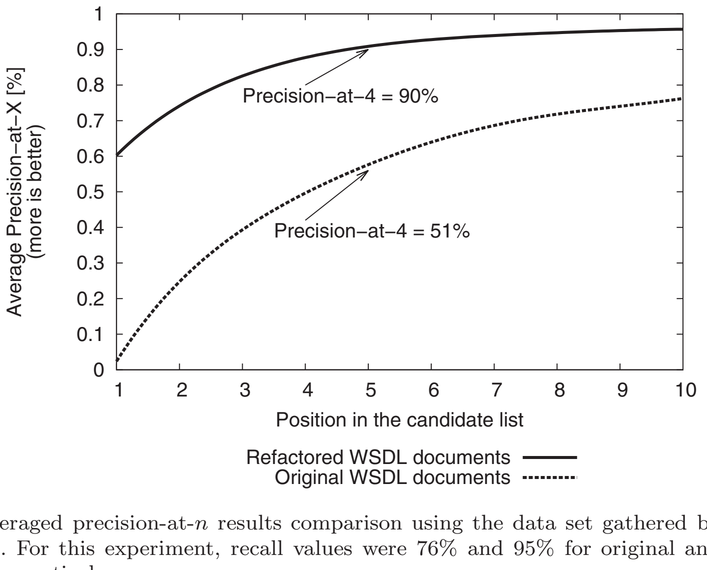


Fig. 14. Averaged precision-at-$n$ results comparison using the data set gathered by Heß *et al.* (see Ref. 22). For this experiment, recall values were 76% and 95% for original and refactored documents, respectively.

documents was more affected by the noise introduced in the registry than the discoverability of the Refactored ones. For instance, Fig. 14 shows that 51% of the Original WSDL documents were ranked at the fourth position at most, whereas for the first experiment this value was 64%, i.e. a fall of 13%. Regarding Refactored WSDL documents, the Precision-at-4 results fell from 93% to 90%, i.e. only 3%. These result may indicate that although more WSDL documents have been published in the registry, refactored ones are still more discoverable.

Both experiments provide empirical evidence showing that employing the proposed refactorings on service implementations improves the discoverability of WSDL documents. Due to the approach to service discovery employed, precision-at-$n$ results can be data-set and query-set specific, and these results can not be generalized to other experimental conditions. As the proposed refactorings rely on re-grouping operations for improving service internal cohesion, remodeling data-types for making them representative of domain objects, and following good naming conventions for making operations and arguments names self-descriptive, it is reasonable to expect at least a small retrieval advantage when applying the refactorings, versus not applying them. This is because the underpinnings of many approaches to service discovery lie in the descriptiveness of WSDL documents. As a result, WSDL documents with representative and semantically related keywords may ameliorate discovery systems retrieval effectiveness.[44]

## 6. Future Research Opportunities

The presented analysis allows bringing together many opportunities for future research. One line relates to code-first service refactorings. First, the refactorings studied in this paper could be automated with the help of an IDE. As a starting point, we will use IntelliJ Idea,[f] a Java IDE that has many built-in refactoring functions. Similarly, the development of IDE-based tools based on ad-hoc service refactorings aimed at removing the root causes of discovery anti-patterns have also proved to benefit service retrieval effectiveness. We have recently proposed an approach following this idea,[29] which also includes an ad-hoc WSDL generation tool that deals with some anti-patterns usually caused by existing WSDL generation tools. However, refactoring effort and developer adoption are negatively affected since in this paper we rely on known refactorings and widely-used WSDL generation tools (i.e. Axis). Then, a study quantifying the trade-off "discovery effectiveness versus refactoring effort/adoptability" of both approaches is underway.

Second, the relationships between the anti-patterns and other OO metric catalogs could in turn lead to investigate the effects of other kind of early refactorings on anti-patterns and service discovery. This includes considering traditional metrics such as the ones proposed by Halstead, McCabe, and Henry and Kafura,[57] and eventually newer ones.[2]

[f] Intellij Idea, http://www.jetbrains.com/idea.

Third, we will incorporate into our analysis less popular WSDL generation tools such as EasyWSDL and JBoss' *wsprovide*. The goal is bringing our findings to a broader audience. Note that the tool used to generate WSDL documents from Java has some incidence in the quality of WSDL specifications in terms of anti-pattern occurrences. We have made some interesting progress towards assessing this incidence in Ref. 36. It is worth noting that Ref. 36 still focuses on anti-pattern occurrences, without paying attention to service discovery performance. This is because tools such as EasyWSDL and WSProvide are used by a small fraction of Web Service developers compared to Axis. Moreover, this latter dominates in open environments where Web Service registries are needed, as opposed to intranet Web Services, where registries are just replaced by (much smaller) service catalogs.

A fourth line of research includes the design and development of version support techniques to allow consumers that use old WSDL documents versions to continue using the refactored WSDL documents until they migrate to the improved versions. In this sense, several researchers have proposed techniques to extend Web Services standards with version support.[24] Similarly, Becker *et al.*[3] addressed the problem from the perspective of evolving service interfaces.

Fifth, comparing the discoverability effectiveness of two, or more, WSDL documents is still an open problem. Although the anti-patterns identified in Ref. 44 allow developers to compare some essential aspects of WSDL documents, it does not enable a fair quantitative comparison in terms of retrieval effectiveness. Indeed, toolkits for testing standard-complaint discovery algorithms have been proposed in the past.[35] These toolkits comprise a set of functions to simplify the benchmarking process, such as generating a collection of synthetically generated WSDL documents and queries, automatically. This allows researchers to build benchmarks comprising a publicly available training data-set of real world Web Services, a publicly available test data-set, a list of all services in the training data-set relevant to each query of the test data-set, and a reproducible evaluation methodology. We believe such benchmarks promote more and better research in the field, in particular to improve the discoverability of Web Services. This idea can be exploited to effectively quantify the trade-off mentioned above.

Sixth, recent works have put the focus on exploiting the behavioral — and not only structural or textual — properties of services interfaces.[25] For example, the sequencing of operations messages, the relationships among any incoming and/or outgoing messages, and whether operations can be partially executed or not. We plan to investigate whether the artifacts used for describing behavioral aspects of services interfaces have measurable properties, and in turn whether these measurable properties are related to independent source code metrics and/or discovery effectiveness. We will particularly focus on this from the perspective of service selection,[19] which is the problem of choosing a particular service for inclusion into a client application from a relevant service list after a registry has been queried. As a starting point, we will base our research on the *observability testing metric* from component-based software development,[18] which is the idea of reasoning about a component/service operational behavior by analyzing the expected input and output data, and how data is transformed. This is analogous to distinguishing the allowed functional data mappings performed by a service and therefore its behavior. We have built a prototype service selection method able to build a Test Suite representing the analyzed behavior of a service, which can be then exercised by client applications to further improve discovery.

Finally, there is a recent trend toward using REST services.[49] Unlike WSDL described services, which rely on SOAP and hence XML for being accessed from client applications, REST services use a simpler and more lightweight communication stack given by HTTP plus JSON. This simplicity and lightweightness has motivated the use of REST Web Services for providing services to mobile applications, so that energy is saved. In this context, there are languages analogous to WSDL being developed for describing REST services, such as the Web application description language (WADL). In fact, we have recently proposed algorithms for discovering REST services,[46] which provides a keyword based interface to search over an index of WADL documents preprocessed via text mining and clustering techniques. Currently, we are evaluating this support using 1400 REST descriptions extracted from the Mashape repository,[g] which opened up the opportunity of studying REST-specific bad specification practices. Therefore, we are building a catalog similar to that of WSDL anti-patterns (with six bad practices already identified). Based on this, we will propose refactoring actions and evaluate their impact on WADL discoverability using our registry.[46]

## 7. Conclusion

WSDL documents play a very important role in enabling third-party consumers to understand, discover and reuse Web Services.[9] It has been shown that these requirements can be fulfilled provided some common WSDL anti-patterns are avoided/corrected when deriving WSDL documents by applying specific refactorings. However, an inherent prerequisite for applying these guidelines is that services are built in a contract-fist manner, by which developers have more control on the WSDL of their services. However, the industry mostly relies on code-first Web Service development, which means that developers first build service implementations and then generate the corresponding service contracts from the code.

We have focused on how to obtain WSDL documents that minimize anti-pattern occurrences when using code-first. Based on the idea that some quality attributes of a software can be predicted during development time, we worked on the hypothesis that anti-pattern occurrences can be avoided by considering OO metrics values from the code implementing services. We used well-established statistical methods to identify the set of OO metrics that best correlate and explain anti-pattern occurrences by using a data-set of real Web Services. We also studied the effect of

[g]https://www.mashape.com/.

applying simple metric-driven code refactorings to the Web Services of the dataset on the anti-patterns in the generated WSDLs. Interestingly, we found that these code refactorings, which are very easy to apply by developers, effectively reduce anti-patterns and thus improve service contracts. In addition, we quantified the effect of removing or mitigating anti-patterns based on the abovementioned source code refactorings on service discovery. As a result, we empirically confirmed that removing anti-patterns or at least reducing the number of their occurrences by following this approach allows refactored services to be better ranked during discovery. This increases service chances of being discovered and, in turn, consumed.

## Acknowledgments

We thank the anonymous reviewers for their constructive comments to improve the paper. We acknowledge the financial support provided by ANPCyT through grant PICT-2012-0045.

## References

1. E. Agichtein, E. Brill, S. Dumais and R. Ragno, Learning user interaction models for predicting web search result preferences, in *29th Annual Int. ACM SIGIR Conf. Research and Development in Information Retrieval* (*SIGIR '06*), Seattle, Washington, USA, pp. 3–10 (ACM Press, NY, 2006).
2. J. Al Dallal, Measuring the discriminative power of object-oriented class cohesion metrics, *IEEE Trans. Softw. Eng.* **37** (2011) 788–804.
3. K. Becker, J. Pruyne, S. Singhal, A. Lopes and D. Milojicic, Automatic determination of compatibility in evolving services, *Int. J. Web Serv. Res.* **8** (2011) 21–40.
4. M. Berón, H. Bernardis, E. Miranda, D. Riesco, M. J. A. Pereira and P. Henriques, Wsdlud: A metric to measure the understanding degree of wsdl descriptions, in *Symp. Languages*, *Applications and Technologies* (*SLATe' 2015*) (Universidade Complutense de Madrid, 2015).
5. M. B. Blake and M. F. Nowlan, Taming Web Services from the wild, *IEEE Internet Comput.* **12** (2008) 62–69.
6. M. Campbell-Kelly, The rise, fall, and resurrection of software as a service, *Commun. ACM* **52**(5) (2009) 28–30.
7. S. Chidamber and C. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Softw. Eng.* **20**(6) (1994) 476–493.
8. Z. Cong and A. Fernandez, Semantic web services in agreement technologies, in *Agreement Technologies*, ed. S. Ossowski, Governance and Technology Series, Vol. 8 (Springer, Netherlands, 2013), pp. 137–148.
9. M. Crasso, J. M. Rodriguez, A. Zunino and M. Campo, Revising WSDL documents: Why and how, *IEEE Internet Comput.* **14**(5) (2010) 30–38.
10. M. Crasso, A. Zunino and M. Campo, Combining query-by-example and query expansion for simplifying Web service discovery, *Inf. Syst. Front.* **13** (2011) 407–428.
11. M. Crasso, A. Zunino and M. Campo, A survey of approaches to Web service discovery in service-oriented architectures, *J. Database Manage.* **22**(1) (2011) 103–134.
12. X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes and J. Zhang, Simlarity search for Web services, in *31st International Conference on Very Large Data Bases* (*VLDB

*2004*), eds. M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley and K. B. Schiefer (Morgan Kaufmann, Toronto, Canada, 2004), pp. 372–383.
13. M. O. Elish, A. H. Al-Yafei and M. Al-Mulhem, Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of eclipse, *Adv. Eng. Softw.* **42**(10) (2011) 852–859.
14. J. Erickson and K. Siau, Web service, service-oriented computing, and service-oriented architecture: Separating hype from reality, *J. Database Manag.* **19**(3) (2008) 42–54.
15. T. Erl, *SOA Principles of Service Design* (Prentice Hall. 2007).
16. J. Fan and S. Kambhampati, A snapshot of public Web Services, *SIGMOD Record* **34**(1) (2005) 24–32.
17. N. E. Fenton and S. L. Pfleeger, *Software Metrics*: *A Rigorous and Practical Approach*, 2nd edn. (PWS Publishing Co., Boston, 1998).
18. A. Flores and M. Polo, Testing-based process for component substitutability, *Softw. Test. Verif. Reliab.* **22**(8) (2012) 529–561.
19. M. Garriga, A. Flores, C. Mateos, A. Zunino and A. Cechich, Service selection based on a practical interface assessment scheme, *Int. J. Web Grid Serv.* **9**(4) (2013) 369–393.
20. T. Gyimothy, R. Ferenc and I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, *IEEE Trans. Softw. Eng.* **31**(10) (2005) 897–910.
21. R. Harrison, S. J. Counsell and R. V. Nithi, An evaluation of the mood set of object-oriented software metrics, *IEEE Trans. Softw. Eng.* **24**(6) (1998) 491–496.
22. A. Heß, E. Johnston and N. Kushmerick, ASSAM: A tool for semi-automatically annotating semantic Web Services, in *Int. Semantic Web Conf.*, Lecture Notes in Computer Science, Vol. 3298, pp. 320–334.
23. X. Jiang and Y. Li, Web service matching based on natural semantic annotation, *Inf. Technol. J.* **12** (2013) 857–861.
24. M. B. Juric, A. Sasa, B. Brumen and I. Rozman, WSDL and UDDI extensions for version support in Web Services, *J. Syst. Softw.* **82**(8) (2009) 1326–1343.
25. B. Laxmaiah, G. S. Reddy, L. S. Shankar, L. C. Sekhar and A. D. Kumar, Behavior evolution of Web Services with dynamic adaptation, *Int. J. Comput. Trends Technol.* **4**(1) (2013) 13–22.
26. S.-H. Li, S.-M. Huang, D. C. Yen and C.-C. Chang, Migrating legacy information systems to web services architecture, *J. Database Manag.* **18**(4) (2007) 1–25.
27. C. Mateos, M. Crasso, J. M. Rodriguez, A. Zunino and M. Campo, Measuring the impact of the approach to migration in the quality of web service interfaces, *Enterp. Inf. Syst.* **9**(1) (2015) 58–85.
28. C. Mateos, M. Crasso, A. Zunino and J. L. Ordiales Coscia, Detecting WSDL bad practices in code-first Web Services, *Int. J. Web Grid Serv.* **7** (2011) 357–387.
29. C. Mateos, J. M. Rodriguez and A. Zunino, A tool to improve code-first web services discoverability through text mining techniques, *Softw.*, *Prac. Exp.* **45**(7) (2015) 925–948.
30. C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu and Q. Xie, Exemplar: A source code search engine for finding highly relevant applications, *IEEE Trans. Softw. Eng.* **38**(5) (2012) 1069–1087.
31. P. Meirelles, C. Santos, J. Miranda, F. Kon, A. Terceiro and C. Chavez, A study of the relationships between source code metrics and attractiveness in free software projects, in *Brazilian Symp. Software Engineering* (*SBES '10*), pp. 11–20, Los Alamitos, CA, USA (IEEE Computer Society, 2010).

32. G. A. Miller, Wordnet: A lexical database for english, *Commun. ACM* **38**(11) (1995) 39–41.
33. H. R. Motahari Nezhad, G. Y. Xu and B. Benatallah, Protocol-aware matching of web service interfaces for adapter development, in *19th Int. Conf. World Wide Web* (*WWW'10*), pp. 731–740, New York, NY, USA (ACM, 2010).
34. S. Murad, Using semantic services in service-oriented information systems, *IEEE Potentials* **32**(1) (2013) 36–46.
35. S.-C. Oh and D. Lee, Wsben: A Web Services discovery and composition benchmark toolkit, *Int. J. Web Serv. Res.* **6**(1) (2009) 1–19.
36. J. L. Ordiales Coscia, C. Mateos, M. Crasso and A. Zunino, Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools, *Int. J. Web Grid Serv.* **9**(2) (2013) 107–126.
37. M. Papazoglou and W.-J. Heuvel, Service oriented architectures: Approaches, technologies and research issues, *VLDB J.* **16**(3) (2007) 389–415.
38. M. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, Service-oriented computing: State of the art and research challenges, *Computer* **40**(11) (2007) 38–45.
39. M. Papazoglou and W.-J. van den Heuvel, Service-oriented design and development methodology, *Int. J. Web Eng. Technol.* **2**(4) (2006) 412–442.
40. J. Pasley, Avoid XML schema wildcards for Web Service interfaces, *IEEE Internet Comput.* **10** (2006) 72–79.
41. M. Perepletchikov, C. Ryan, K. Frampton and H. W. Schmidt, Formalising service-oriented design, *J. Softw.* **3**(2) (2008) 1–14.
42. D. Radjenović, M. Heričko, R. Torkar and A. Živkovič, Software fault prediction metrics: A systematic literature review, *Inf. Softw. Technol.* **55**(8) (2013) 1397–1418.
43. J. M. Rodriguez, M. Crasso and A. Zunino, An approach for web service discoverability anti-pattern detection, Vol. 12 (Rinton Press, Incorporated, Paramus, NJ, 2013), pp. 131–158.
44. J. M. Rodriguez, M. Crasso, A. Zunino and M. Campo, Improving Web Service descriptions for effective service discovery, *Science of Computer Programming* **75**(11) (2010) 1001–1021.
45. J. M. Rodriguez, C. Mateos and A. Zunino, Assisting developers to build high-quality code-first web service apis, *J. Web Eng.* **14**(3&4) (2015) 251–285.
46. J. M. Rodriguez, A. Zunino, C. Mateos, F. O. Segura and E. Rodriguez, Improving REST service discovery with unsupervised learning techniques, *9th Int. Conf. Complex, Intelligent, and Software Intensive Systems* (Blumenau, Brazil, 2015).
47. D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler and D. Fensel, Web Service modeling ontology, *Appl. Ontol.* **1**(1) (2005) 77–106.
48. G. Salton, A. Wong and C. S. Yang, A vector space model for automatic indexing, *Commun. ACM* **18**(11) (1975) 613–620.
49. Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne and X. Xu, Web services composition: A decade's overview, *Inf. Sci.* **280** (2014) 218–238.
50. K. Sivashanmugam, K. Verma, A. P. Sheth and J. A. Miller, Adding semantics to Web Services standards, in *Int. Conf. Web Services*, pp. 395–401, Las Vegas, NV, USA (CSREA Press, 2003).
51. D. Spinellis, Tool writing: A forgotten art? *IEEE Software* **22** (2005) 9–11.
52. P. Sripairojthikoon and T. Senivongse, Concept-based readability of web services descriptions, in *2013 15th Int. Conf. Advanced Communication Technology* (*ICACT*), pp. 853–858 (IEEE, 2013).

53. P. Sripairojthikoon and T. Senivongse, Concept-based readability measurement and adjustment for web services descriptions, in *16th Int. Conf. Advanced Communication Technology* (*ICACT*), pp. 378–388 (IEEE, 2014).
54. S. Stigler, Fisher and the 5% level, *Chance* **21** (2008) 12–12.
55. E. Stroulia and Y. Wang, Structural and semantic matching for assessing Web Service similarity, *Int. J. Coop. Inf. Syst.* **14**(4) (2005) 407–438.
56. The Apache Software Foundation, Commons-math: The Apache commons mathematics library, 2010, http://commons.apache.org/math.
57. F. F. Tsui and O. Karam, *Essentials of Software Engineering* (Prentice Hall, 2006).
58. R. A. Van Engelen and K. A. Gallivan, The gSOAP toolkit for Web Services and peer-to-peer computing networks, in *2nd IEEE/ACM Int. Symp. Cluster Computing and the Grid* (*CCGRID '02*), pp. 128–135, Washington, DC, USA (IEEE Computer Society, 2002).
59. B. Zeigler and H. Sarjoughian, Service-based software systems, in *Guide to Modeling and Simulation of Systems of Systems*, Simulation Foundations, Methods and Applications (Springer, London, 2013), pp. 205–230.