SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A tool to improve code-first Web Services discoverability through text mining techniques

Cristian Mateos*, Juan Manuel Rodriguez, Alejandro Zunino

ISISTAN Research Institute - UNICEN University
Tandil (B7001BBO), Buenos Aires, Argentina.
Tel./Fax: +54 (249) 443-9682 ext. 35/443-9681
Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

SUMMARY

Service-oriented development is challenging mainly because Web Service developers tend to disregard the importance of the exposed service APIs, which are specified using WSDL documents. Methodologically, WSDL documents can either be manually generated, or inferred from service implementations using WSDL generation tools. The latter option, called code-first, is the most used approach in the industry. However, it is known that there are some bad practices in service implementations or defects in WSDL generation tools that may cause WSDL documents to present WSDL anti-patterns, which in turn compromise the chances of documents of being discovered and understood. In this paper, we present a software tool that assists developers in obtaining WSDL documents with as few WSDL anti-patterns as possible. The tool combines text mining and meta-programming techniques to process service implementations, and is developed as an Eclipse plug-in. An evaluation of the tool by using a data-set of real service implementations in terms of anti-pattern avoidance accuracy, and discovery performance by using classical Information Retrieval metrics –Precision-at-n, Recall and Normalized Discounted Cumulative Gain– is also reported. Copyright © 2013 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: WEB SERVICES; CODE-FIRST; WSDL ANTI-PATTERNS; AUTOMATIC DETECTION; SERVICE DISCOVERY

1. INTRODUCTION

Web Services is the common technological choice for materializing the Service-Oriented Computing (SOC) paradigm [1, 2]. Basically, Web Services enable service providers to implement their services using well-known interoperable Web protocols, such as HTTP or SOAP. In this context, services are offered using sets of atomic operations that are described using the Web Service Description Language (WSDL), which is an XML dialect. Hence, service consumers do not need to know service implementation details because knowing the WSDL document associated to a service is in principle enough for using this latter. In addition, WSDL documents are useful for indexing services in registries so that service consumers can look for the services they need.

Unfortunately, it has been shown in practice that publicly available WSDL documents are usually difficult to understand, and they do not contain useful pieces of information (e.g., ambiguous names) making Web Service registries ineffective [3]. As a result of this, Web Services in open environments are not as wide-spread as expected. Firstly, service consumers have the difficult task of finding the Web Services they need. Secondly, once a service consumer has found a suitable Web Service, they

*Correspondence to: Cristian Mateos, cmateos@conicet.gov.ar have to reason about its functionality in order to use it just by inspecting the associated WSDL document.

For the process of discovering and understanding Web Services from their WSDL documents to be effective, service providers must provide meaningful WSDL documents upon developing services [3]. For the case of *contract-first* Web Services, in which WSDL documents are manually developed before service implementation, by considering a catalog of bad WSDL practices –i.e., WSDL anti-patterns– the discoverability and understandability of WSDL documents is greatly improved [4]. In short, 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 eight anti-patterns considered in this paper are classified into three categories of WSDL bad practices: high-level service interface specification (*Enclosed data model*, *Redundant port-types, Redundant data models* and *Low cohesive operations in the same port-type* anti-patterns), comments and identifiers (*Inappropriate or lacking comments* and *Ambiguous names* anti-patterns) and service message exchange (*Whatever types* and *Undercover fault information within standard messages* anti-patterns).

The most popular approach to build Web Services in the industry is code-first. With this approach, developers first implement a service and then generate the corresponding WSDL by automatically extracting and deriving the interface from the implemented code. This means that WSDL documents are not directly created by developers but are instead automatically derived via language-dependent tools such as Axis' Java2WSDL[†]. Precisely, bad coding practices and defects in generation tools may cause WSDL documents from having WSDL anti-patterns, and hence less discoverable and understandable WSDL documents are produced. With this in mind, the definition and development of an approach and tool support for *avoiding* anti-patterns for the case of code-first Web Services is the motivation of the present work. Our approach consists on detecting issues or bad coding practices in the service code and generation defects that might otherwise map to problematic WSDL documents under existing tools. Due to its high popularity in back-end development, and to limit the scope of our research, we focus our discussion on Web Services implemented in Java.

In addition to the techniques for detecting the bad coding practices, we also briefly present an Eclipse plug-in called Anti-pattern Free Java2WSDL (AF-Java2WSDL) that implements the techniques. As a result, these techniques can be easily and readily integrated into real-life code-first Web Service development. The plug-in analyses Web Service source code, detects possible issues and suggests suitable refactorings to remove them. Furthermore, the plug-in uses its own WSDL generation tool, which avoid anti-patterns –e.g., *Redundant port-types*– that strictly depend on the way service implementations are mapped to WSDL documents.

The sets of experiments performed to evaluate our approach were twofold. First, since some of the studied WSDL anti-patterns are inherently subjective, we evaluated and compared the accuracy levels achieved by our techniques to automatically spotting bad coding practices in service implementations –that in turn may cause anti-patterns– with respect to manual WSDL anti-pattern detection. To this end, a data-set of 60 real service implementations comprising 430 operations was used. Second, we compared the discoverability of WSDL documents generated via our tool and Java2WSDL, the most popular WSDL generation tool for Java. In this case, we employed classical Information Retrieval metrics, particularly Precision-at-*n*, Recall and Normalized Discounted Cumulative Gain (NDCG). Our discovery experiments showed that WSDL documents generated using AF-Java2WSDL outperforms those derived via Java2WSDL, especially in terms of Precision-at-*n* and NDCG.

The rest of the paper is organized as follows. Firstly, Section 2 discusses related work for improving WSDL document understandability and discoverability. Then, Section 3 presents our approach for improving WSDL documents understandability and discoverability in the context of code-first Web Services. Section 4 briefly describes the Eclipse plug-in that materializes our approach. Section 5 presents the experimental evaluation. Finally, Section 6 concludes the paper and presents prospective future works.

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

2. BACKGROUND AND RELATED WORKS

Web Service technology is a rather complex amalgam of well-known Internet technologies, designed to expose software in order to be consumed by other software. Therefore, developing Web Services without automatic assistance might be a difficult error-prone task that requires developers to have a deep knowledge of Web standard protocols and languages. To overtake these issues, there are tools to assist Web Service development in the most common platforms, such as Java or .Net. For instance, Axis2[‡] is a WSDL generation tool for Java and the Web Services Description Language Tool (WSDL.exe)[§] is the out-of-the-box tool for Web Service development in .Net [5]. This type of development tools have been described as essential for encouraging developers to adopt new technologies and methodologies [6].

Despite being useful, these tools do not take into account the whole Web Service development process. This might result in some issues when mapping a system specification, which are critical for enterprise system development [7], into Web Services. In [8], the authors propose a methodology to implement Web Services that might use other Web Services from the intended system Unified Modeling Language (UML) specification. Basically, the methodology consists in five steps. The first one is specifying a system using UML. Then, the second step is discovering which Web Services might replace some of the functionality defined in this first specification. As a result, the developer would obtain an analogous UML-like specification that considers the discovered Web Services. The third step consists in mapping this specification into the system implementation, being this latter automatically obtained from the UML-like specification, and developers must implement the missing logic. The forth step is to generate the WSDL documents for the new Web Services, i.e., the services offered by the new system. Finally, the fifth step is to publish the new WSDL documents in a service registry. It is worth noting that most of the steps, such as discovering Web Services, mapping UML specifications into the system skeleton and WSDL documents [9], and publishing the resulting Web Services, can be accomplished by means of automatic or semi-automatic tools. In [10], the authors propose a similar methodology, but they use the Business Process Execution Language (BPEL) to describe the service composition scheme, which is a language defined for such purpose.

To ease the development of successful service-oriented systems, several domain specific tools and frameworks have been proposed. For instance, there are frameworks for developing Web Service based systems for digital libraries [11], negotiation process management in business-to-business contexts [12], biological research [13], and e-Government systems [14], to name a few. In addition to domain specific problems, all these works address at least one of two problems: Web Service replaceability (i.e., the step two in [8]) and Web Service discoverability. In [12, 11, 14], these problems are addressed by means of defining homogeneous interfaces. The problem of this approach is that all the providers of that kind of systems must agree on using that interface, which in the long-term might difficult adding new features. However, the authors of [13] have acknowledged that Web Services are implemented by different providers that have no agreement on how the Web Service interface should be defined. As a result, the authors have pointed out that Web Services in the biological domain are difficult to discover and use mainly due to their sheer number, the fact that they are distributed, and the low quality of their documentation. Therefore, they propose a catalog for assisting the use of Web Services in this particular domain.

The difficulty of discovering and replacing Web Services is not inherent to a particular domain, but to the Web Services themselves. Previous to Baghat et al. [13], different authors [15, 16, 17, 18, 19] have detected this issue in Web Services. To solve this issue, different researchers [15, 16, 19, 20, 21, 22, 23] have proposed Web Service registries. Basically, this kind of tools can be divided into two types [22]: registries for regular Web Services and registries for Semantic Web Services. The latter are Web Services that have been annotated with semantic information from an ontology. Although this kind of semantic Web Services can be easily discovered, they are rather uncommon

[‡]Axis2: http://axis.apache.org/axis2/java/core/
[§]Web Services Description Language Tool: http://msdn.microsoft.com/en-us/library/7h3ystb6(v=vs.80).aspx

in real life applications because they are difficult to specify. One the other hand, regular (a.k.a. syntactic) Web Services are easier to specify, but as stated at the beginning of this paper they suffer when bad WSDL specification practices are followed, making it difficult to discover and consume these kind of Web Services [17, 13, 3, 4].

Table I. Catalog of Web Service discoverability anti-patterns

Anti-pattern	Symptoms	Manifestation
Enclosed data model	Occurs when the type definitions are placed in WSDL documents rather than in separate XSD ones.	Evident
Redundant port-types	Occurs when several port-types offer the same set of operations.	Evident
Redundant data models	Occurs when many types for representing the same objects of the problem domain coexist in a WSDL document.	Evident
Whatever types	Occurs when a type represents any object of the domain.	Evident
Inappropriate or lacking comments	Occurs when (1) a WSDL document has no comments, or (2) comments are inappropriate and not explanatory.	(1) Evident, or (2) Not immediately apparent
Low cohesive operations in the same port-type	Occurs when port-types have weak semantic cohesion.	Not immediately apparent
Ambiguous names	Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent
Undercover fault information within standard messages	Occurs when output messages are used to notify service errors. Sometimes (1) whatever types are returned and operation comments suggest anti-pattern occurrence. Otherwise (2) it is necessary to analyze service implementation.	(1) Not immediately apparent, or (2) Present in service implementation

Moreover, these bad practices or issues in regular Web Service discoverability mainly stem from the quality of the provided WSDL documents [18, 3]. In a broad sense, evaluating code quality has been always a major concern for software developers [24]. To evaluate Web Service discoverability, which is a quality factor, [4] presents a catalog of *WSDL anti-patterns*. These anti-patterns subsume previously identified Web Service issues, such as name issues [18] and data-type definition issues [25, 26], as well as define new ones. Web Services with anti-patterns usually have less relevant terms making a Web Service registry index ineffective. In addition, using Web Services affected by anti-patterns is difficult because they might have unclear documentation and unrepresentative names. This catalog describes eight common anti-patterns, which are summarized in Table I, and affect Web Service discoverability. Furthermore, [27] shows the importance of taking into account these anti-patterns during real-life SOC development for a large enterprise.

Taking into account these anti-patterns, the authors have proposed several methodologies for developing Web Services. Firstly, the authors propose several algorithms and heuristics for automatically detecting some of the anti-patterns presented in [4]. These algorithms and heuristics, from now on called "anti-pattern detectors", are designed for analyzing WSDL documents. In [28], these algorithms were extended to support all the anti-patterns and to improve the performance of the previously defined heuristics. This work is suitable for assisting *contract-first* Web Service development, which requires writing the WSDL document first and then implementing the Web Service logic. Despite the benefits of these heuristics for developing Web Services, *code-first* Web Service development is the usual way of implementing Web Services in the industry, which means that developers do not manually write the WSDL documents. Yet, these detectors might be useful to assess automatically generated WSDL documents from the point of view of the heuristics they rely on.

The mentioned heuristics were integrated into an Eclipse plug-in to assist developers in following different guidelines for developing, publishing, discovering and consuming Web Services [29]. The main advantage of this tool over the other previously mentioned is that it was designed to assist traditional developers to consume and provide Web Services without the need to specify the system using any extra modeling language. In addition to assisting Web Service development by means of the heuristics, this tool assists developers that need to use Web Services for implementing their systems. To do this, the tool is integrated with the Web Service Query By Example (WSQBE) service registry [19]. When using WSQBE, developers do not explicitly write a query to discover services, but instead they define an expected service interface using Java interfaces. Then, the registry generates a query automatically and returns a list of candidate Web Services. The other major feature of this tool is a component that semi-automatically creates the glue code between the selected Web Service and the expected interface. The main advantage of using this glue code is that the system is independent of the used Web Service, and if in the future the developer decides to replace the Web Service to consume, he/she only needs to regenerate the glue code. According to the authors, the generated glue code only adds a small performance overhead when calling services.

Regarding code-first Web Service development, a recent study [30] has pointed out that there are statistical correlations between some classical OO source code quality metrics –mainly those from the Chidamber and Kemerer's catalog such as Abstract Type Count and Coupling Between Objects– and the presence of anti-patterns in generated WSDL documents. Therefore, applying refactorings to improve these metrics might result in better quality WSDL documents. However, this might not be always the case because some of the relevant correlations ($p - value < 0.05$) are as low as 0.6. Therefore, this indirect approach to anti-pattern removal does not cover all anti-patterns effectively. Another limitation of the approach is that these refactorings focus on modifying structural aspects of service codes only, such as changing POJOs by primitive data-types and viceversa, splitting a service into several smaller services, adding/removing parameters, and so on. However, these refactorings do not cover certain service coding bad practices that lead to textual (but not necessarily structural) discoverability and understandability problems in generated WSDL documents. Examples are absence or inappropriate documentation and meaningless/ambiguous parameter and method names.

Finally, [31] proposes a tool for assisting the migration of COBOL legacy systems to Web Service technologies minimizing the impact of the WSDL anti-patterns in the resulting Web Services. In this context, this tool might be considered as a code-first tool because it takes as input the implementation and helps the developer to migrate it to SOA. However, the goal of this tool is to assist developers to modify the glue code that exposes a COBOL program as a Web Service rather than generating that glue code as Web Services. To perform this task, the tool relies on standard tools, such as Axis2 or WDSL.exe.

This paper analyses the possibility of adapting some of the techniques initially presented in [32], as well as presenting new ones, for detecting possible issues in the Web Service source code that could be mapped into anti-patterns in the generated WSDL document. In addition, we present an anti-pattern aware technique for mapping the resulting service logic code into the WSDL document that describes the implemented service. Compared to [30], our approach attempts to cover more WSDL anti-patterns, whereas as opposed to [29], we target code-first (and not contract-first) Web Services.

The novelty of this work is that, unlike [32, 29, 28], this approach is designed for code-first Web Service development, and specifically addresses anti-pattern root causes, which is not the case in previous work [30]. In this context, it is worth noticing that [32, 29, 28] detect anti-patterns by inspecting WSDL documents only, while the proposed approach detects coding practices in Java that could be mapped into WSDL discovery anti-patterns when Java code is mapped into WSDL documents by means of WSDL generation tools. This practice is precisely promoted by code-first. Although thoughtfully designed contract-first WSDL documents have been proven to be affected by less anti-patterns than code-first WSDL documents, even when using anti-pattern aware tools [31], code-first Web Service development is widely used because is cheaper and provide faster time to market [27]. In addition, developing contract-first Web Services requires experts to write WSDL documents, instead of generating them automatically. Therefore, our approach does not aim at generating anti-pattern free WSDL documents, but at achieving a balance between WSDL document quality and the benefits of code-first Web Service development in terms of faster service construction times.

3. THE ANTI-PATTERN FREE JAVA2WSDL (AF-JAVA2WSDL) TOOL

Basically, code-first Web Service development consists in writing the logic of a Web Service in a programming language and then using tools for generating the WSDL documents, which expose this logic as Web Services. A previous work [30] suggests that code-first Web Service implementations affect the quality of the resulting Web Services in terms of WSDL anti-pattern occurrences. Moreover, in code-first Web Services, the anti-patterns might stem from two non mutually exclusive sources: the service implementation or the WSDL document generation tool. For instance, if there are no comments in a service implementation, the WSDL generation tool has no comments to place in the generated WSDL document (i.e., implementation flaws). In contrast, defining a port-type more than once is a result of how the WSDL document is generated (i.e., WSDL generation tool defects). Therefore, our approach consists in assisting code-first Web Service developers to write good quality service code and then to correctly map this latter into WSDL documents to prevent the introduction of as many anti-patterns as possible.

Basically, the first part of our approach focuses on removing the anti-patterns that arise from bad service implementation practices. These anti-patterns are *Inappropriate or lacking comments*, *Ambiguous names*, *Low cohesive operations in the same port-type*, *Whatever types* and *Undercover fault information within standard messages*. Essentially, our approach consists in analyzing the classes that will be exposed as services looking for possible issues that could translate into WSDL anti-patterns. In order to perform the analysis, detection heuristics, or *detectors* for short, have been designed for each one of these anti-pattern. These detectors are explained in detail in Section 3.1.

The second part consists in correctly mapping the classes that are to be exposed as Web Services into good WSDL documents. This is done through a *mapper* component that takes into account all the names and comments that the developer places in the source code of the service implementation. In addition, the mapper was designed to avoid WSDL generation related anti-patterns, such as the *Enclosed data model* anti-pattern or the *Redundant port-types* anti-pattern. Lastly, the mapper also minimizes the number of repeated types (i.e., the *Redundant data models* anti-pattern) and correctly using the WSDL fault mechanism. All in all, the mapper deals with potential anti-patterns that are not caused by bad service coding practices, but defects in current WSDL generation tools. This mapper is explained in Section 3.2.

3.1. Anti-pattern detection heuristics (detectors)

As suggested earlier, the detectors represent a set of heuristics that analyzes source code to spot potential WSDL anti-patterns causes. The following paragraphs explain the mechanics associated to the detectors of each anti-pattern.

To avoid the *Inappropriate or lacking comments anti-pattern*, the detector firstly verifies that all methods in the exposing class have comments, and when some method lacks comments a problem is informed to the developer. Then, the detector analyses the existing comments and compares them with the method name and method parameter names. To do this, our detector firstly creates a phrase joining the method name with the parameter names, which are separated by commas, using the "with" word. To obtain individual words, a simple word splitting technique is used, which assumes camel casing as the input format. For instance, for the method *setStockValue* that receives *stockName* and *newValue* as parameters the resulting phrase is "set stock value **with** stock name**,** new value". Then, this sentence is compared with the first sentence of the comments. Notice that camel casing is the de facto coding standard in Java, and thus this assumption is valid.

To compare two phrases, the detector generates two trees that represent each phrase semantics. The first step determines whether each word in the phrase is a verb or a noun, which is done using a probabilistic context free grammar (PCFG) parser [33]. Then, hypernyms for each word in its noun/verb role are extracted from WordNet [34]. This is done to prone cases in which a word might be a verb or a noun, e.g., "fax" or "mail". As a result, each word has one or several hypernyms which indicates what they are in a taxonomy. For instance, "mail" as verb might have several meanings, one of them is that **mail** is a **conveyance, transport** which is a **instrumentality, instrumentation** and so on to finally be an **entity**. The detector uses these hypermyns to create the tree that represents the phrase. Firstly, it creates a tree with an artificial root and iteratively adds the hypermyns to that tree. Algorithm 1 shows the detailed steps for adding hypernyms. Then, the similitude between two

Algorithm 1 Hypernym addition to a tree.

```
1: procedure ADDHYPERNYMS($tree, hypernymList$)
2:     if ISEMPTY($hypernymList$) then
3:         RETURN()
4:     end if
5:     $child \leftarrow hypernymList.first$
6:     if ISNOTCHILDOF($child, tree$) then
7:         $newChild \leftarrow$ CREATENODE(child)
8:         ADDNEWNODE($tree, newChild$)
9:     end if
10:    $nextTree \leftarrow$ GETEQUALNODE(tree,child)
11:    COMBINETREE($nextTree, hypernymList.rest$)
12: end procedure
```

trees is calculated as follows:

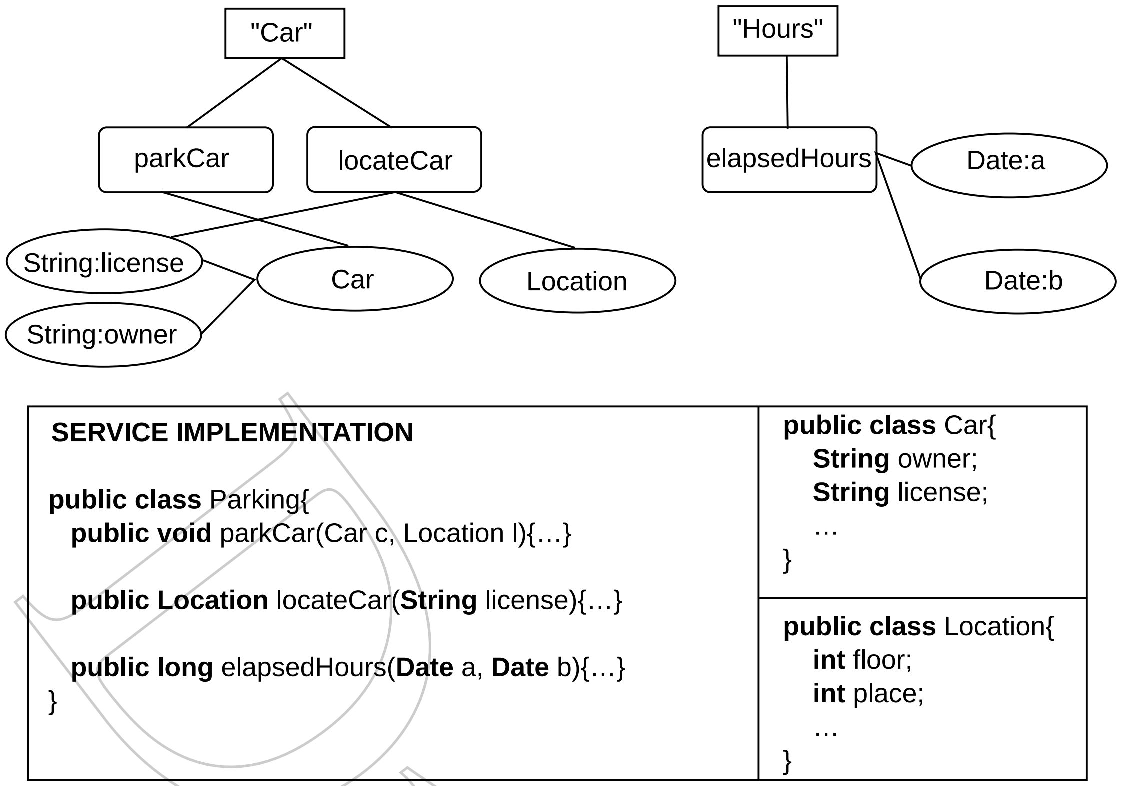
$$sim(tree_1, tree_2) = \frac{depth(deepestShareSubTree(tree_1, tree_2))}{max(depth(tree_1, tree_2))}$$

According to our preliminary analysis, a similitude higher than 10% is enough to assure that the comments and its operations are related [28].

For detecting *Ambiguous names* anti-pattern causes, the detector follows a three-step heuristic. Firstly, it verifies whether the names of classes, attributes and method arguments have an appropriate length, which should fall between 3 and 30 characters [4]. Then, it verifies that the names do not use unrepresentative names, namely *param, arg, var, obj, object, foo, input, output, in#, out# and str#*, where # might be replaced by a number or nothing. Then, the third step varies for methods and parameters names. For methods names, the heuristic add at the beginning of it the string "it must" and then it applies the PCFG parser to determine whether the resulting string has a verb and a non-pronoun noun. This is because a method name should have the form "verb+noun". The detector analyses this sentence word by word and not the whole sentence because the parser tends to misclassify nouns for verbs when the sentence is malformed. Finally, if the name under analysis belongs to a method, it applies the PCFG parser to the whole name to check whether it is a noun or a phrase.

The detectors for the causes of both *Inappropriate or lacking comments* anti-pattern and *Ambiguous names* anti-pattern are fairly similar to the heuristic proposed for detecting these anti-patterns in WSDL documents in [28]. This is because the problem is very similar and only adaptation from WSDL to Java language were required to implement them.

To detect the *Low cohesive operations in the same port-type* anti-pattern causes, the detector creates an undirected graph that represents the relations of a service methods [35]. Unlike the two



```
public class Parking{
  public void parkCar(Car c, Location l){…}

  public Location locateCar(String license){…}

  public long elapsedHours(Date a, Date b){…}
}
```

```
public class Car{
   String owner;
   String license;
   …
}
```

```
public class Location{
   int floor;
   int place;
   …
}
```

Figure 1. Methods relationship graph

previous detectors, this is completely different from the one described in [28] because preexistent literature suggests that graph based methods are fairly good to measure cohesion in source code compared to text-only methods [35]. The graph has three kinds of edges:

- Methods: this kind of edges represents the methods of the class to be analyzed.
- Types: these edges are the types used by the methods and all referenced types. However, primitive types and types too generic, such as java.lang.Object or java.lang.String in Java, are complemented with the variable name because they are common to many languages. In the case of Java, types are considered as too generic if they belong to the packages java.lang.* or java.utils.*.
- The singular nouns in the methods name: these edges are the singular form of the nouns, which are detected using the PCFG parser, in the methods names.

On the other hand, the arcs on the graph represent one or more of the following relationships:

- has an attribute of the type: this is a relationship between two types. It represents a type having an attribute of another type. This arises as a consequence of XSD reuse constructs.
- receives as parameter: this relationship represents a method that receives as input a particular type.
- returns: this relationship represents a method that returns a particular type.
- parametrized with: this relationship exists when a type is parametrized with another type. It also represents relationships through ignored types, such as arrays or maps.
- has the noun: this relationship exists between a method and all the nouns in its name.

Once the graph including the methods of a port-type has been generated, the detector determined whether the graph is connected or not. Otherwise, the detector selects the largest connected subgraph and reports the methods that have no edges in that subgraph as uncohesive. For instance, Figure 1 depicts a class implementation with three methods: two for parking lot management and one for determining the difference between two dates, namely elapsedHours. In this example, the latter method is detected as unrelated to the service that the Parking class implements because it is not connected to the largest subgraph.

To detect possible *Whatever types* anti-pattern causes, the detector uses a list of classes that are known to be too generic [36]. When a method uses one of these classes, the detector informs that there is an issue. The classes considered too generic are Object, Vector, List, Map, Collection, Enumeration, Vector<Object>, List<Object>, Map<Object, Object>, Collection<Object> and Enumeration<Object>. All these classes belong to java.lang.* or java.utils.*.

Finally, detecting the *Undercover fault information within standard messages* anti-pattern causes involves analyzing the structure of a method output. Firstly, the detector verifies whether a method has exceptions defined. If this is not the case, the detector analyses the output class field names looking for the following keywords: "ping", "error", "errors", "fault", "faults", "fail", "fails", "exception", "exceptions", "overflow", "mistake", "misplay". These names often indicate that the output conveys error information that should be returned using an exception instead of placing the information in the return value [4].

This two last anti-pattern cause detectors are similar to the ones presented in [28] since both use syntactic analysis to detect potential issues. However, the structures that they compare are very different because manifestation of these issues in Java is very different from their manifestation in WSDL. For instance, the tag `<any>` is the manifestation of the *Whatever-types* anti-patterns in WSDL documents, but in Java, the cause of this anti-pattern can be the use of the class `Object` or not using generics when available, e.g., using `List` instead of `List<Clazz>` [36]. Something similar happens with the *Undercover fault information within standard messages* anti-pattern. This is because when detecting the anti-pattern in WSDL documents, the detector uses the presence of fault messages in an operation to rule out the anti-pattern. Alternatively, when analyzing the anti-pattern cause in Java, the detector rules this out by checking whether a method declares that it might throw an Exception.

Notice that this part of the approach cannot tackle all the existing anti-patterns. For example, if the WSDL document generation tool disregards the comments in the source code, the generated WSDL documents would be affected by the *Inappropriate or lacking of comments* anti-pattern regardless the class was commented or not. Then, it is necessary to use a tool that follows WSDL documents good practices, such as the one described in the following section, to complement the detectors.

3.2. *The Java to WSDL custom mapper*

We have developed a Java to WSDL mapper that receives as input a Web Service implementation and returns the WSDL document that describes it and an XSD file which defines the required types. Basically, the Web Service implementation can be a Java interface with one or more realizing classes, or a Java class that defines both the interface and the behavior. Currently, our generation tool only maps the abstract part of the WSDL document (i.e., it ignores the *binding* and *service* tags), but it can be easily extended to generate WSDL documents ready to be deployed.

The mapper creates a WSDL document that exposes the Java Interface/Class, or for brevity sake, class from now on. Firstly, the mapping defines a port-type for the class that would be exposed as a Web Service. Since only one port-type is defined per class, there is no possibility that the mapping process generates a WSDL document affected by the *Redundant port-types* anti-pattern. Then, the mapper creates an operation for each public method of the input class. The mapper comments and names each operation with the comments and the name associated to the mapped method. When several methods share the same name, the tool renames the operation with the method name concatenated to a number since WSDL does not support method overloading.

For each mapped method, the mapper defines an input and an output message. In addition, the mapper generates as many fault messages as exceptions that might be thrown by the method. The mapper assumes that the class methods' inputs and outputs are Plain Old Java Objects (POJOs) that only have properties accessible through getters/setters. This is because operation inputs and outputs are expected to convey data, but not behavior. In contrast, method faults are expected to extend from java.lang.Exception, which is the mechanism provided by Java to raise a fault. In both cases, the generation tool creates the XSD file by generating XSD complex types to represent the classes, and adding recursively the required types to express all the necessary classes. The recursion ends when the generation tool finds a primitive type, such as int, double or float, or a class which maps to an XSD built-in type, such as java.lang.String, java.util.List or java.util.Map. Notice that complex types are named after the class they represent and each of its properties are named after the attribute or field they map to.

Inputs and outputs classes are mapped into XSD types to prevent the generation of redundant data-types, i.e., the *Redundant data models* anti-pattern. This is because it is assumed that the input and output of each operations are represented by POJOs, and in a good object-oriented design there should not be two POJO classes with the same information. However, if the input/output classes are not POJOs, there might be redundant data-types. For instance, when a class inherits from another class, it might not add new attributes to the class, but behavior. Since operations in Web Services only receive and return data as input/output, the two classes would be mapped into two XSD types with the same attributes. However, it is of no use using inheritance to add or modify in classes that are intended as input/output in Web Services' operations.

To illustrate the WSDL document generation process, let us consider the construction of a service that returns the forecast for a city in a particular country. Firstly, the developer should implement the service (we are ignoring here the step that checks the source code for implementation flaws). The following codes are fragments of the service implementation:

```
//Input Class
public class CityInfo {
        private String cityName;
        private String province;
        //Getters and Setters
        ....
}
//Output Class
public class ForecastInfo {
        private double temperature;
        private double relativeHumidity;
        //more data
        ....
        //Getters and Setters
        ....
}
//Service Implementation
public class Forecast {
        /**
         * Returns the forecast information for the day in a city.
         */
        public ForecastInfo getWeatherForecast(CityInfo place){
                //Service logic
        }
}
```

For this code, the generation tool creates the following WSDL document, which has been simplified for the sake of exemplification. Since the *Enclosed data model* anti-pattern impact depends on whether the type will be reused or not, we decided to enclose them in this example. However, the tool offers the user the possibility of generating them into a separated XSD file.

```
<definitions ...>
  <types>
    <xsd:schema ...>
      <xsd:element name="getWeatherForecastRequest">
        <xsd:complexType>
```

```
          <xsd:sequence>
            <xsd:element name="place" ... type="CityInfo"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      ...
      <xsd:complexType name="CityInfo">
        <xsd:sequence>
          <xsd:element name="cityName" type="xsd:string" />
          <xsd:element name="province" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
     ...
    </xsd:schema>
  </types>
  <message name="getWeatherForecastRequest">
    <part name="parameters" element="xsd1:getWeatherForecastRequest" />
  </message>
  ...
  <portType name="Forecast">
    <operation name="getWeatherForecast">
      <documentation>Returns the forecast information for the day in a
          particular city.</documentation>
      <input message="tns:getWeatherForecastRequest" />
      <output message="tns:getWeatherForecastResponse" />
    </operation>
  </portType>
</definitions>
```

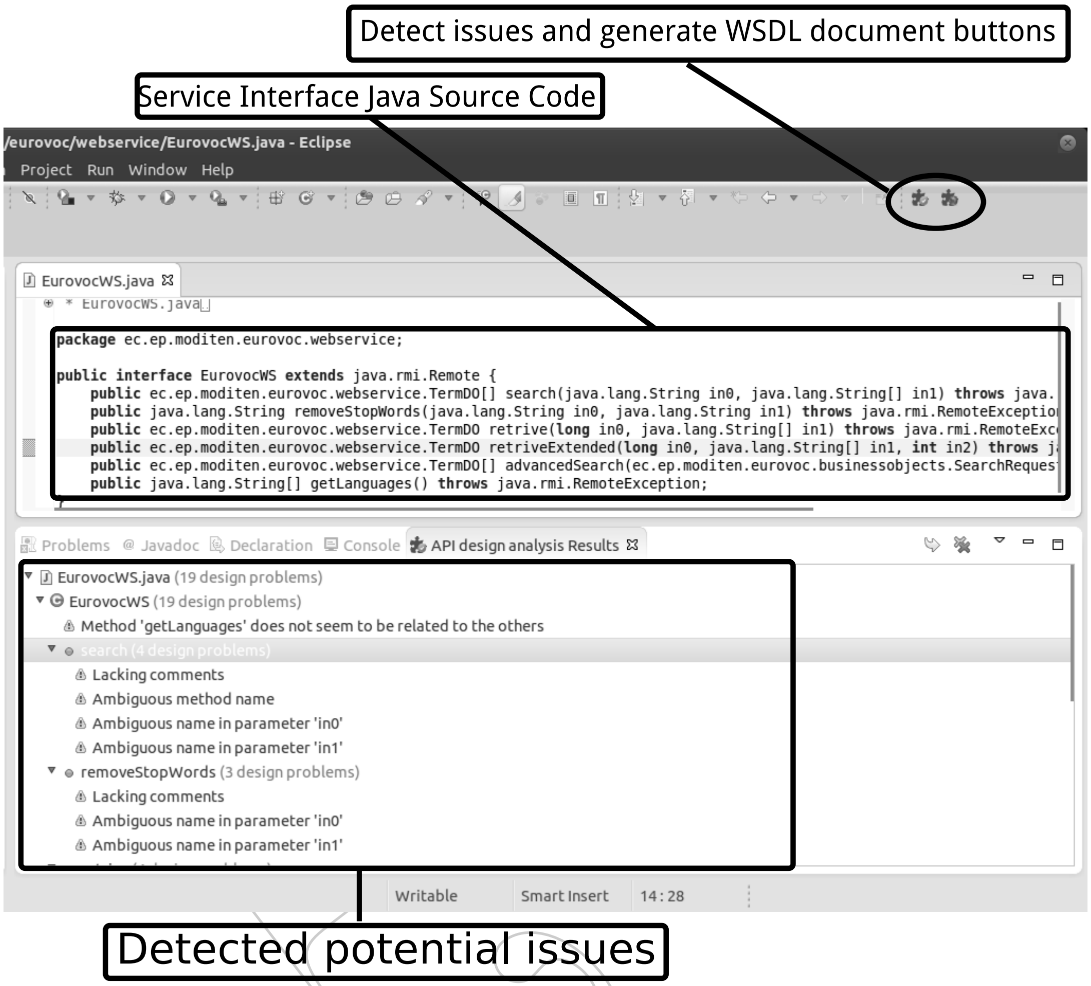
Notice that all the names in the class as well as the comments have been preserved. However, since the WSDL file defines elements that cannot be directly matched to Java elements, such as "message", some unrepresentative names were added. This is the main drawback of our Java to WSDL mapper. Yet, our mapper as well as several other similar tools, such as Axis Java2WSDL, try to preserve as much semantic information as possible, which is good for service discovery [4].

4. THE ECLIPSE PLUG-IN

Our detectors and our Java to WSDL mapper have been implemented in the form of a plug-in for Eclipse, which can be downloaded from the EasySOC project homepage[¶]. Basically, the plug-in has two main features: detecting potential issues in the Java code of a Web service and mapping that code into WSDL documents. Service developers can use both features in two ways: through the main toolbar or through pop-up menus. To analyze the class currently being edited, developers can use the button in the toolbar. However, if developers want to analyze more than one class at once, they should select the classes in the Eclipse Package Explorer, which by default is placed in the left part of the screen, and click on "Analyze API design" in the pop-up menu. In contrast, the button in the toolbar as well as the item in the pop-up menu named "Generate WSDL document" only take into account the Java class that is currently being edited.

The Java code analyzer uses the Abstract Syntactic Tree (AST) API, which is a model of a Java program in the form of Java classes provided by Eclipse. By working with the AST instead of the source code, we ensure that the plug-in is compatible with code from different Java versions, e.g, Java 1.4, Java 5 and Java 6. To facilitate adding new heuristics or modifying the existing ones, we have defined an interface that receives a Java AST and returns a list of potential discoverability issues. Then, each heuristic is implemented by extending that interface. Therefore, the rest of the plug-in code is independent of which heuristics are used and how they are implemented. Figure 2

[¶]EasySOC: https://sites.google.com/site/easysoc/


Detect issues and generate WSDL document buttons
Service Interface Java Source Code
/eurovoc/webservice/EurovocWS.java - Eclipse
Project Run Window Help
EurovocWS.java
Problems @ Javadoc Declaration Console API design analysis Results
Writable Smart Insert 14 : 28
Detected potential issues

```
 * EurovocWS.java

package ec.ep.moditen.eurovoc.webservice;

public interface EurovocWS extends java.rmi.Remote {
    public ec.ep.moditen.eurovoc.webservice.TermDO[] search(java.lang.String in0, java.lang.String[] in1) throws java.
    public java.lang.String removeStopWords(java.lang.String in0, java.lang.String in1) throws java.rmi.RemoteExceptio
    public ec.ep.moditen.eurovoc.webservice.TermDO retrive(long in0, java.lang.String[] in1) throws java.rmi.RemoteExc
    public ec.ep.moditen.eurovoc.webservice.TermDO retriveExtended(long in0, java.lang.String[] in1, int in2) throws j
    public ec.ep.moditen.eurovoc.webservice.TermDO[] advancedSearch(ec.ep.moditen.eurovoc.businessobjects.SearchReques
    public java.lang.String[] getLanguages() throws java.rmi.RemoteException;
```

```
▼ EurovocWS.java (19 design problems)
  ▼ EurovocWS (19 design problems)
      Method 'getLanguages' does not seem to be related to the others
    ▼ search (4 design problems)
        Lacking comments
        Ambiguous method name
        Ambiguous name in parameter 'in0'
        Ambiguous name in parameter 'in1'
    ▼ removeStopWords (3 design problems)
        Lacking comments
        Ambiguous name in parameter 'in0'
        Ambiguous name in parameter 'in1'
```

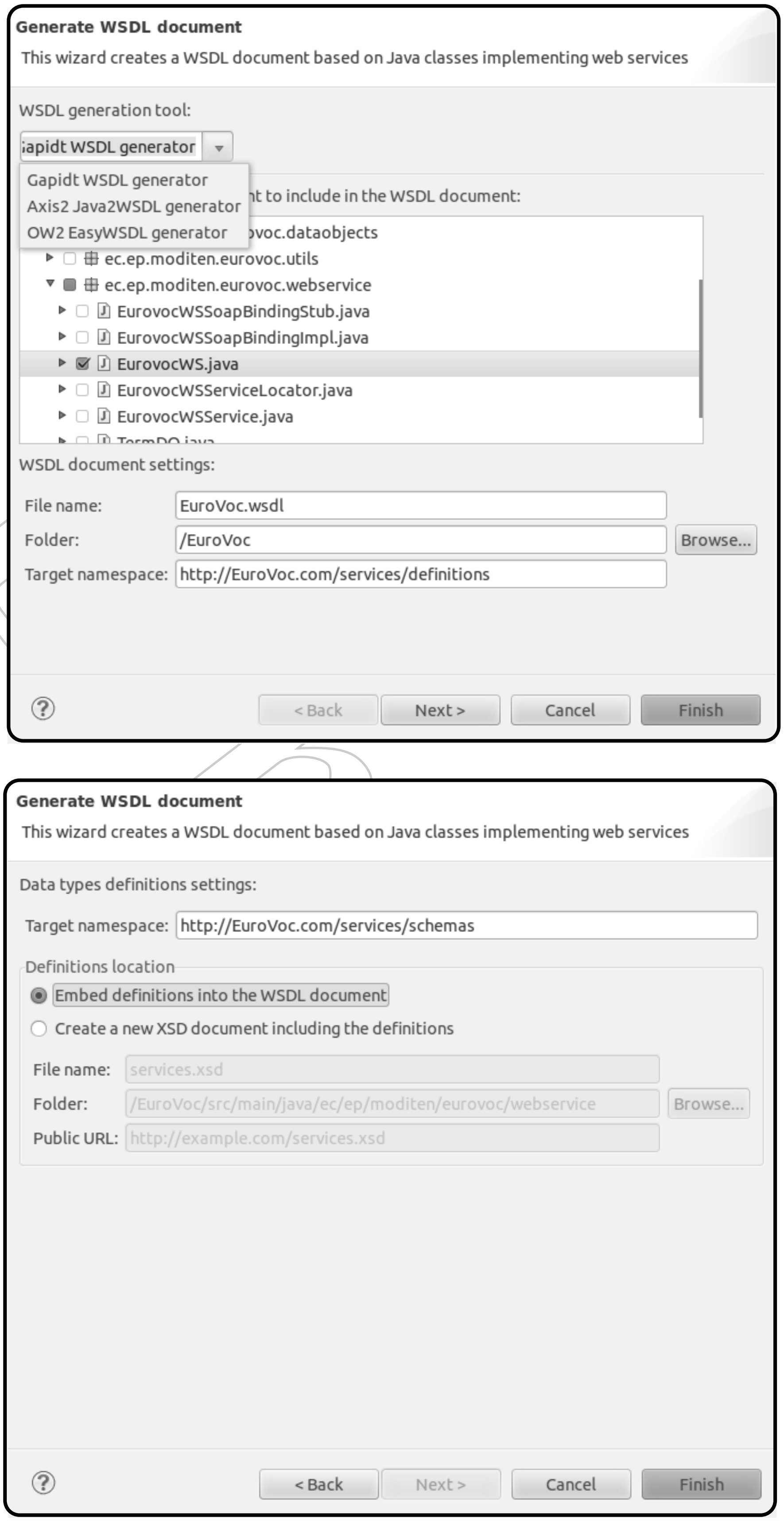
Figure 2. Eclipse plug-in

illustrates how developers can access both features of the plug-in using the toolbar buttons. In addition, this figure also shows how the plug-in reports possible issues in a class.

Regarding mapping classes to WSDL documents, the plug-in offers three options: Axis2 Java2WSDL, EasyWSDL[‖] and naturally our generator (see Section 3.2). If the developer chooses to use the standard tools, i.e., Axis2 Java2WSDL or EasyWSDL, the plug-in WSDL document generation wizard collects some parameters used by the tools and then delegates the generation task to them. Otherwise, the plug-in uses our mapping methodology. The mapping functionality is also implemented using Eclipse AST. The implementation consists in adding the classes as port-types, and their methods as operations taking into account their names, comments and input/output/fault classes. Finally, the classes used as input/output/fault are mapped into XSD types. The developer can choose whether the XSD types are exported into a different file or embedded in the generated WSDL document.

Figure 3 depicts the two screens of the WSDL generation wizard. In the first screen, the developer selects which class to expose as a Web Service and, therefore, that should be mapped to a WSDL document. In addition, the developer can select which WSDL document generation tool they want to use and where the file will be generated. This screen also gives the developer the option of generating the WSDL document or going to the second screen to configure the XSD file generation

[‖]EasyWSDL: http://easywsdl.ow2.org/



Generate WSDL document

This wizard creates a WSDL document based on Java classes implementing web services

WSDL generation tool:

Gapidt WSDL generator

- Gapidt WSDL generator
- Axis2 Java2WSDL generator
- OW2 EasyWSDL generator

nt to include in the WSDL document:

- ovoc.dataobjects
- ec.ep.moditen.eurovoc.utils
- ec.ep.moditen.eurovoc.webservice
  - EurovocWSSoapBindingStub.java
  - EurovocWSSoapBindingImpl.java
  - ☑ EurovocWS.java
  - EurovocWSServiceLocator.java
  - EurovocWSService.java
  - TermDO.java

WSDL document settings:

File name: EuroVoc.wsdl

Folder: /EuroVoc

Target namespace: http://EuroVoc.com/services/definitions

Generate WSDL document

This wizard creates a WSDL document based on Java classes implementing web services

Data types definitions settings:

Target namespace: http://EuroVoc.com/services/schemas

Definitions location

● Embed definitions into the WSDL document

○ Create a new XSD document including the definitions

File name: services.xsd

Folder: /EuroVoc/src/main/java/ec/ep/moditen/eurovoc/webservice

Public URL: http://example.com/services.xsd

Figure 3. WSDL document generation wizard

Detector	Analyzed parts	Number of parts
Inappropriate or lacking comments	All public methods	568
Ambiguous names (method names)	English named public methods	556
Ambiguous names (parameters names)	English named parameters of public methods	986
Low cohesive operations in the same port-type	All public methods parameters and return values	508
Whatever types	All public methods of classes/interfaces with more than one method	1,563
Undercover fault information within standard messages	All public methods	568

Table II. Detectors: Analyzed parts per service class

parameter. The second screen allows the user to select the namespace of the XSD file, and when targeting our tool, it allows the developer to select whether to generate the XSD type definitions within the WSDL document or into a separated file for reusing them.

5. EVALUATION

To evaluate AF-Java2WSDL, we used a Java project data-set that comprises 60 services. This data-set is a subset of the data-set presented in [30], i.e., the ones that could be fully compiled and built. An important feature of this data-set is that the services came from real-life projects that actually implement Web Services or EJBs (Enterprise Java Bean) that were servified. These projects were gathered from different sources, particularly Google Code, Exemplar [37] and Merobase (http://merobase.com). Notice that unlike the data-sets used for evaluating our previous anti-pattern remotion tools [32, 29, 28], the data-sets in this paper do not consist of WSDL documents only. This is because the former approaches were designed for directly working with the WSDL documents as contract-first suggests. In contrast, the tool in this work operates at the service implementation level, which implies that the WSDL documents are not handled directly but indirectly through modifying source code, as code-first suggests. Finally, the Java project data-set built was complemented with two WSDL data-sets from the literature [38, 39] to increase the size of the experiment in terms of the amount of WSDL documents used in the evaluation regarding discoverability (see Section 5.2).

We performed two separated experiments. The first one was designed to measure the effectiveness of our detectors for automatically spotting potential discoverability issue causes. This was done by employing classical confusion matrices. Second, we performed an experiment to assess the discoverability of the documents generated with AF-Java2WSDL when compared to a well-known code-first Web Service development tool, namely Java2WSDL, which is part of Apache Axis2 (http://axis.apache.org/axis2/java/core/).

5.1. Anti-pattern detectors accuracy

We have manually analyzed each class that is mapped to a Web Service looking for anti-pattern causes. To avoid possible biases, the manual analysis was carried out by one final-year System Engineering student, who had taken the "Service-Oriented Computing" course[**] of the Systems Engineering BSc. program at the UNICEN during 2012. As such, the student had knowledge

[**]SOC Course UNICEN: http://www.exa.unicen.edu.ar/~cmateos/cos

Detector	**Automatic**	**Manual**		**Total**	**Overall Accuracy**	**False Negative**	**False Positive**
		Negative	**Positive**				
Inappropriate or lacking comments	**Negative**	175	0	175	97.53%	0%	3.56%
	Positive	14	379	393			
Ambiguous names	**Negative**	1,456	8	1,464	98.83%	0.55%	12.82%
	Positive	10	68	78			
Low cohesive operations in the same port-type	**Negative**	430	4	434	96.45%	0.92%	18.91%
	Positive	14	60	74			
Whatever types	**Negative**	1,499	0	1,499	100%	0%	0%
	Positive	0	64	64			
Undercover fault information within standard messages	**Negative**	567	0	567	100%	0%	0%
	Positive	0	1	1			

Table III. Detectors: confusion matrices

on Service Oriented Computing concepts, code-first Web Service development and WSDL anti-patterns. It is worth noting that the student did not have knowledge on the internals of our plug-in nor its detection heuristics, i.e., he/she played the role of a regular (but not expert) plug-in user.

Since different anti-patterns affect different parts in the Java classes, the number of analyzed parts for each anti-pattern varies. For example, to assess the *Inappropriate or lacking comments* anti-pattern causes detector effectiveness, from the data-set, 568 parts were analyzed, but 1,563 parts were analyzed for detecting the *Whatever types* anti-pattern causes. Table II shows which parts are analyzed by each detector and the quantity of them in the employed data-set.

The results from the manual analysis were compared with the results of executing the detection heuristics in the projects. Using this information, we constructed a confusion matrix for each detector (two for *Inappropriate or lacking comments* and *Ambiguous names*). Table III presents the different confusion matrices as well as the detection heuristics accuracies, false positive, and false negative rates. This matrix was filled using the analysis performed by a third party developer to avoid biases. It is worth noting that the evaluation of the *Inappropriate or lacking comments* detector has been divided into two experiments: lacking comments detection and inappropriate comments detection. Also, the *Ambiguous names* detector experiment has been divided into two parts. This is because the heuristics for analyzing method names and parameter names are slightly different. Therefore, the accuracy of each part of these heuristic might be different from each other. In consequence, we present an analysis for each of these detectors' parts in Table IV.

On average, considering only the experiments for the main five detectors (i.e., those from Table III), the detectors have an accuracy of 98.56% being the worst accuracy 96.45%. In addition, the average false negative and false positive rates among all gray rows were 0.29% and 8.83%, respectively. In all the cases, the accuracy was never lower than 96%. Moreover, the false positives rates were high in some cases, which might mean that the heuristics are too sensitive to some problems inherent to real-life source code.

Firstly, the detector of *Inappropriate or lacking comments* presented a 97.53% accuracy with a low false positive rate. In a further analysis, it can be seen that the accuracy and low false positive rates are due to the Lacking comments detection heuristic, which worked flawless in the experiment.

Detector	**Automatic**	**Manual**		**Total**	**Overall Accuracy**	**False Negative**	**False Positive**
		Negative	**Positive**				
Lacking comments	**Negative**	233	0	233	100%	0%	0%
	Positive	0	335	335			
Inappropriate comments	**Negative**	175	0	175	93.99%	0%	24.13%
	Positive	14	44	58			
Ambiguous names (method names)	**Negative**	504	4	508	97.66%	0.79%	18.75%
	Positive	9	39	48			
Ambiguous names (parameters names)	**Negative**	952	4	956	99.49%	0.42%	3.33%
	Positive	1	29	30			

Table IV. Detectors' parts: confusion matrices

Yet, the Inappropriate comments detection heuristic performed somewhat worse (see Table IV). It had an accuracy of 93.99% and a high rate of false positives. However, a main issue is that developers tend to not comment classes and interfaces that are intended to be exposed as APIs [40].

In the case of the detector for *Ambiguous names*, the analysis shows that the effectiveness of this heuristic is heavily impacted by the performance of the ambiguous method name heuristic. Yet, this heuristic worked fairly well both for method and argument names (see Table IV). In addition, the heuristic for analyzing whether a parameter name is ambiguous worked almost flawlessly.

The detector for *Low cohesive operations in the same port-type* anti-pattern causes consists on determining if all the methods of a class or interface are related from a functional perspective. The accuracy of this detector was higher than 96%. However, the false positives are as high as 24.13%, which means that the developer should decide whether the anti-pattern is actually affecting the class. On the other hand, the highest false negatives rate is 0.92%, suggesting that the detectors rarely missed an anti-pattern.

The results for the detector of *Whatever types* anti-pattern causes were flawless, which stem from the fact that there are only a few practices for using too generic types/classes. Finally, there were no cases of the possible root causes of the *Undercover fault information within standard messages* anti-pattern. As a result, the evaluation of the detector is not complete because we only can say that it did not produced false positive results. Yet, the fact that there were no cases means that the developers, at least in the analyzed data-set, tended to use correctly the Java error handling mechanism. Therefore, if the WSDL documents generated from this data-set were affected by the *Undercover fault information within standard messages* anti-pattern, it the problem would be caused by the generation tools and not by the developers.

5.2. *WSDL discovery performance*

We conducted experiments to measure the implications on discovery of early removing anti-patterns in service implementations and using our mapper to generate WSDL documents. This was done by performing some refactorings to consider the detections made by AF-Java2WSDL to avoid anti-patterns, generating associated WSDL documents, and then discovering these documents. The goal is to analyze whether placing effort on refactoring service implementations, as suggested by AF-Java2WSDL, could increase the chance of an improved service of being discovered or not.

To do this, we executed the automatic detection heuristics using as input the main service classes from the above data-set. Then, a developer not having knowledge on the internals of our heuristics refactored these classes with the help of our plug-in to remove the reported issues. Table V describes the specific refactoring actions used for each detected issue. Notice that ideally the plug-in should not detect any issue in the refactored classes. However, in a 4.44% of the cases, the plug-in still reported *Inappropriate or lacking comments* due to the detector false positives. In addition, there were two classes in which their source code was so complex and cryptic that the developer was unable to completely refactor these service implementations.

Detector	**Refactoring**
Lacking comments	Add comments to the method.
Inappropriate Comments	Rewrite the comments to make them more descriptive.
Ambiguous Names	Replace ambiguous names with non-ambiguous ones.
Low cohesive operations in the same port-type	Remove uncohesive operations from the class.
Whatever types	Replace type `java.lang.Object` for more specific ones.
Undercover fault information within standard messages	Modify the return class so as to not conveying error information and define an exception that might by be thrown by the method.

Table V. Applied refactorings

Methodologically, the evaluation consisted of three steps. In a first step, a corpus of code-first WSDL documents were grouped into two groups. One group called "AF-Java2WSDL" contained those WSDL documents generated after using AF-Java2WSDL to refactor the 60 service implementations of our data-set and generate their WSDLs with our mapper. Another group had the original service implementations [30], whose WSDL documents were built by using Axis Java2WSDL. Second, we supplied a service registry with both groups of WSDL documents. In addition to these two groups of WSDL documents, four scenarios were derived, each one additionally populating an extra WSDL data-set into the registry:

- Scenario 1: 393 extra publicly available WSDL documents extracted from the data-set described in [38], which is well-known in the area,
- Scenario 2: 1,000 extra random WSDL documents extracted from the data-set described in [39], which comprises 1,664 publicly available WSDL documents in total,
- Scenario 3: 1,664 extra WSDL documents, i.e., the entire data-set described in [39], and
- Scenario 3: 2,057 extra WSDL documents from joining the data-sets in [38] and [39].

Clearly, the purpose of considering these scenarios was to assess discovery performance as the size of the registry grows.

As a third step, under each scenario, we queried the employed registry using one query per available service operation in the services of the data-set (i.e., 430 queries). 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 5 documents are all relevant to a query and the next 5 are all non-relevant, we have a precision of 100% at a cut-off of 5 documents but a precision of 50% at a cut-off of 10 documents. Finally, we averaged the results over the total number of queries.

As the reader could observe, the WSDL documents generated via AF-Java2WSDL were basically WSDLs whose associated service implementations were first modified to take into account not some, but all the refactorings suggested by AF-Java2WSDL. The reason behind this decision was that, even when some anti-patterns affect service discovery more than others, they all impact on the discoverability of WSDL documents [4]. Besides, in practice, modifying implementations by considering all the suggestions is not expensive since all of them can be easily performed directly from the Eclipse IDE.

For the experimentation, a publicly available registry implementation of the approach to service discovery presented in [19] was employed. The registry can be downloaded from http://sites.google.com/site/easysoc/home/service-registry . This registry exploits relevant information contained in WSDL documents and, optionally, UDDI entries associated with a service. 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 representation, called Vector Space Model [41], is borrowed from the Information Retrieval area. With this model, documents are seen as collection of terms, whereas each dimension of the space corresponds to a separate term (single words). 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 [19]. This registry returns an ordered list of candidate WSDL documents, sorted according to how similar to the query the associated services are.

Indeed, there are many discovery registries exploiting the Vector Space Model [22], e.g. (see [15] for a pioneer work in this line). However, one of the main differences between the registry used in the experiments and similar alternatives is that the former uses a two-step matching process that greatly reduces discovery time and it is suitable for distributed environments such as WANs or the Internet, where the number of services is large. In this way, our aim was to chose a registry offering a realistic discovery scenario. As an aside, the original anti-pattern paper [4] evaluated the WSDL anti-patterns impact using different textual Web Services registries, and concluded that the impact of the anti-patterns is not registry specific. In other words, anti-patterns negatively affect IR registries alike.

For the sake of fairness we built the queries from the source code of original service implementations. This assumption is analogous to the Query-By-Example concept presented in [19], which means that most of the time users looking for a particular Web Service operation do not use keywords other than the ones that potentially appear in the operation name (i.e., they tend to disregard argument names and documentation). 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, 430 queries were built, one per offered operation. Finally, we associated two WSDL documents with each query, one document belonging to the Axis Java2WSDL group, while another from the AF-Java2WSDL one. For the association we arbitrarily selected the WSDL documents containing the operation needed.

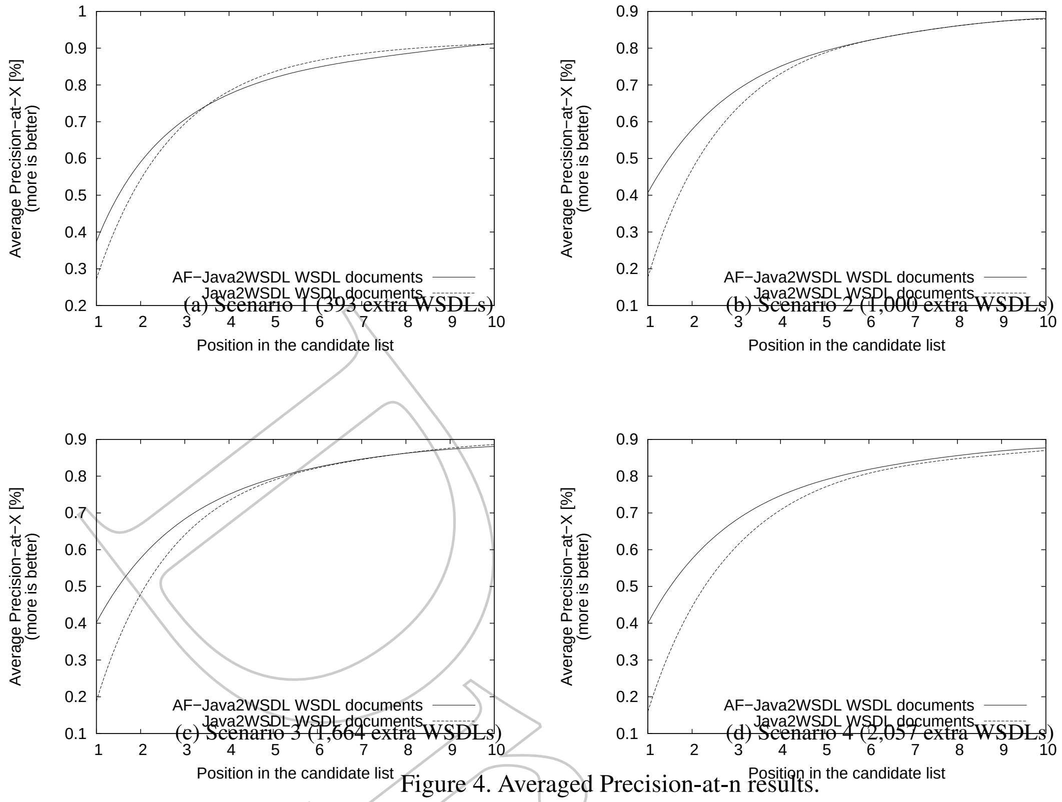
The Precision-at-n results have been calculated for each query with n in [1,10]. We have chosen this window size because we want a good balance between the number of candidates and the number of relevant candidates retrieved. Moreover, we believe that a developer can easily examine 10 Web Service descriptions. Therefore, by setting $n = 10$, we refer to the actual number of relevant services up to only 10 candidates in the result list. Besides calculating Precision-at-n, Recall and Discounted Cumulative Gain (DCG) measures have been calculated. Formally, Recall is defined as:

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

In particular, in our experiments the numerator (*Relevant*) of the Recall 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.

The DCG is a measure for ranking quality and measures the usefulness (gain) of an item based on its relevance and position in the provided list. The higher the DCG is, the better the ranked list is. Formally, DCG is defined as:

$$DCG = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2 i}$$



(a) Scenario 1 (393 extra WSDLs)
(b) Scenario 2 (1,000 extra WSDLs)
(c) Scenario 3 (1,664 extra WSDLs)
(d) Scenario 4 (2,057 extra WSDLs)

Figure 4. Averaged Precision-at-n results.

where p is the size of the candidate list, which for these experiments is 10, and 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).

Figure 4 depicts the averaged Precision-at-*n* results for the 430 queries by smoothing these results using Bézier curves. Results show that, in general, AF-Java2WSDL WSDL documents were ranked before their Axis Java2WSDL counterparts. Particularly, having a higher Precision-at-*1* means that a relevant service was retrieved at the top of the result list. Furthermore, for each scenario, Precision-at-*1* for AF-Java2WSDL and Axis Java2WSDL WSDL documents was:

- Scenario 1: 37.4% and 27.2%, respectively.
- Scenario 2: 40.7% and 17.7%, respectively.
- Scenario 3: 40.2% and 18.8%, respectively.
- Scenario 4: 40.0% and 16.0 %, respectively.

In other words, irrespective of the scenario, the WSDL documents of services whose implementations had been refactored and generated by our tool, were ranked in the first position in the 37.4%-40.0% of the cases. On the other hand, Precision-at-*1* for Axis Java2WSDL documents was in the range of 16.0%-27.2%. Accordingly, discoverers would have to analyze fewer candidates until finding a relevant service when employing AF-Java2WSDL WSDL documents. These results have a great impact on discoverability because users tend to select higher ranked search results [42]. 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% [42].

Moreover, Table VI summarizes the obtained results with respect to Recall and NDCG. On one hand, there were not significant differences as recall was very similar. Likewise, we calculated the

Noise data-set	**# of WSDLs**	Recall		NDCG	
		Axis Java2WSDL	AF-Java2WSDL	Axis Java2WSDL	AF-Java2WSDL
Heß et al.	393	0.9116	0.9116	0.7761	0.7862
Al-Masri & Mahmoud (partial)	1,000	0.8790	0.8813	0.7267	0.7519
Al-Masri & Mahmoud (complete)	1,664	0.8860	0.8813	0.7330	0.7522
Heß et al. + Al-Masri & Mahmoud	2,057	0.8697	0.8767	0.7051	0.7493

Table VI. Performed experiments: Recall and NDCG (with documentation)

DCG for each query and then we averaged the values for the 463 queries. Accordingly, the NDCG was in the range of 70.5%-77.6% and 74.9%-78.6% for the Axis Java2WSDL and AF-Java2WSDL WSDL documents, respectively. Note that these NDCG values provide a global perspective of the performance obtained, which confirms that the refactored WSDL documents surpassed the Axis Java2WSDL ones, obtaining gains of up to 6%.

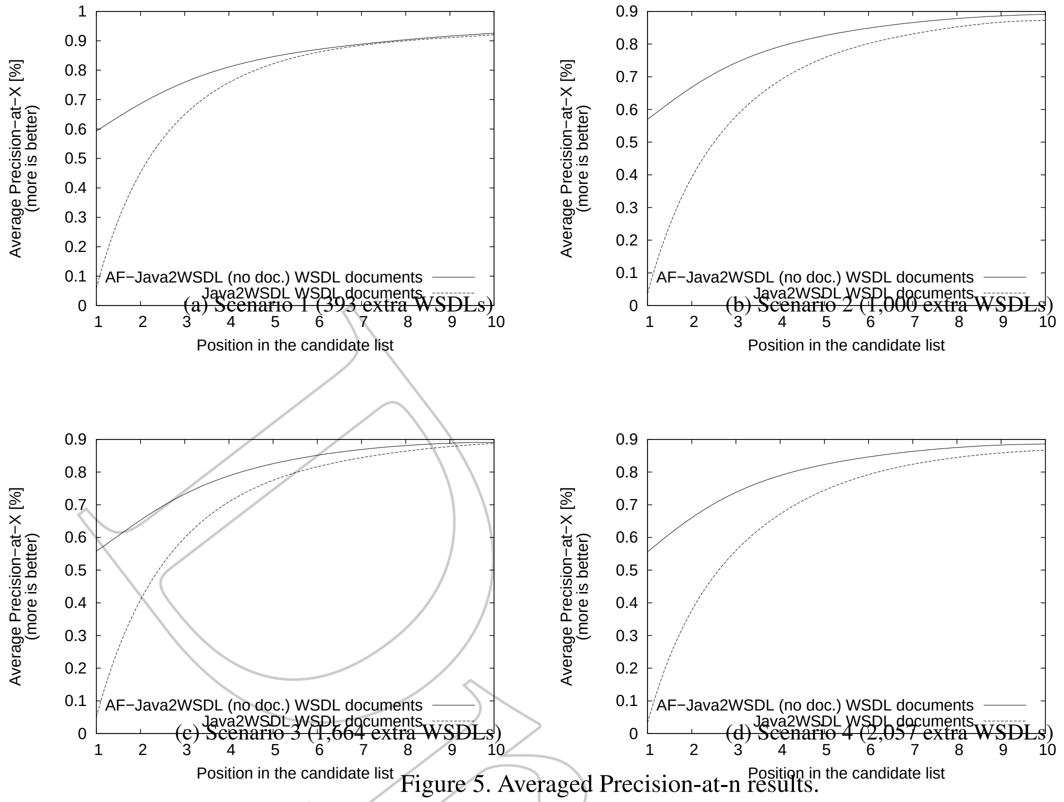
As shown in Figure 4, Precision-at-*n* was always higher in favor of AF-Java2WSDL (or at least equal) for Scenarios 2, 3 and 4, where larger data-sets of extra WSDL documents where used. However, despite having better precision up to $n = 3$, Precision-at-*i* for AF-Java2WSDL documents with $i = 4, 5, 6, 7, 8, 9$ incurred in an overhead of 1-2% compared to that of Axis Java2WSDL documents. In looking for an explanation to this anomaly, we set forth the hypothesis that following all the suggestions made by our tool when generating WSDL documents adds substantial differences to the way documents are indexed in the registry. Particularly, refactoring to avoid the *Innapropriate or lacking comments* anti-pattern results in AF-Java2WSDL WSDL documents having much more terms (because of the added <documentation> tags) that are not present in Axis Java2WSDL documents. This latter fact is since Axis Java2WSDL does not preserve source code comments upon generating WSDL documents. Therefore, when jointly populated in the same syntactic registry, small queries such as the ones used to query the registry may yield very different results.

In this line, we decided to perform a second experiment by reproducing the same steps described up to now but by considering AF-Java2WSDL WSDL documents having the *Innapropriate or lacking comments* anti-pattern. Figure 5 depicts the Precision-at-*n* results for this experiment. Again, the 463 Precision-at-*n* results have been averaged and smoothed using Bézier curves. The tendency of firstly ranking AF-Java2WSDL WSDL documents maintained and was even stronger compared to the previous experiment. Moreover, for each scenario, Precision-at-*1* for AF-Java2WSDL and Axis Java2WSDL WSDL documents was:

- Scenario 1: 59.3% and 6.0%, respectively.
- Scenario 2: 57.0% and 3.7%, respectively.
- Scenario 3: 55.8% and 5.1%, respectively.
- Scenario 4: 55.6% and 3.3%, respectively.

Table VII summarizes the results of Recall and NDCG. On one hand, there were some slight differences w.r.t recall in favor of AF-Java2WSDL documents, with gains in the range of [0.5-2.0]%. Furthermore, AF-Java2WSDL documents were better in terms of NDCG, achieving gains of [5.6-14.3]%.

One possible misinterpretation of these results is that it is always convenient to ignore the *Innapropriate or lacking comments* anti-pattern when using AF-Java2WSDL, since less programming effort is needed and better Precision-at-*n*, Recall and NDCG is obtained. It is worth noting that, from a user's perspective, finding a needed Web Service in a registry involves two steps, namely ranking WSDL documents according to the user's query (automatic) and inspecting the



(a) Scenario 1 (393 extra WSDLs)
(b) Scenario 2 (1,000 extra WSDLs)
(c) Scenario 3 (1,664 extra WSDLs)
(d) Scenario 4 (2,057 extra WSDLs)

Figure 5. Averaged Precision-at-n results.

Noise data-set	**# of WSDLs**	Recall		NDCG	
		Axis Java2WSDL	AF-Java2WSDL	Axis Java2WSDL	AF-Java2WSDL
Heß et al.	393	0.9209	0.9255	0.7654	0.8086
Al-Masri & Mahmoud (partial)	1,000	0.8720	0.8906	0.6982	0.7855
Al-Masri & Mahmoud (complete)	1,664	0.8883	0.8906	0.7141	0.7763
Heß et al. + Al-Masri & Mahmoud	2,057	0.8674	0.8860	0.6835	0.7817

Table VII. Performed experiments: Recall and NDCG (without documentation)

top *n* results (manual). When relevant documents are not retrieved at the topmost positions, good documentation helps in quickly discarding services until reaching a relevant one. Thus, even when not providing any documentation may benefit the ranking step, this trade-off the effort users have to put into the inspection step. Exploring the amount of intra-WSDL documentation that should be provided so as to optimally balance this trade-off is an open problem, and is subject of further research.

To sum up, both experiments evidence that employing the refactorings supported by our tool on service implementations improves the discoverability of WSDL documents. Note that, however, results can be data-set and query-set specific, and hence they cannot be generalized to other experimental conditions. Considering these refactorings aim at reducing WSDL anti-patterns in documents, it is reasonable to expect at least a small retrieval advantage when applying the refactorings, versus not applying them. This is because the performance of many approaches to service discovery depend on the descriptiveness of WSDL documents. Then, WSDL documents with representative and non-ambiguous names, comments and data-types, contribute to improve retrieval effectiveness [4].

6. CONCLUSIONS AND FUTURE WORK

This work proposed a tool that assists code-first Web Service developers to improve the understandability and discoverability of their code-first Web Services. Basically, the tool is intended to detect problems in Web Services implementation code that might map to WSDL anti-patterns in the generated WSDL documents. In addition, the tool includes a custom Java to WSDL mapper that also overcomes some WSDL generation defects present in similar existing mappers. Knowing the potential issues, developers can assess WSDL documents quality and apply suitable refactorings as needed, and thus generating better quality WSDL documents for his/her code-first Web Services.

AF-Java2WSDL enforces practices that have been proved to be necessary for service-oriented systems success [4]. It is already known that when these practices are not followed during the first stages of Web Service development, reingeneering a system afterwards might be costly [27]. Since it seems that SOC developers often disregard these practices [4, 30, 27], a tool for assisting them is necessary for improving a SOC system chance of success.

Unlike the case of the contract-first plug-in previously proposed [32, 29, 28], correcting possible WSDL anti-pattern causes in the Web Service source code does not guarantee that the generated WSDL document will be free of anti-patterns, but it causes the generated WSDL documents to be less affected by anti-patterns, minimizing their negative impact. For instance, if the WSDL generation tools disregard the comments in the source code, the generated WSDL documents will suffer from the *Inappropriate or lacking comment* anti-pattern. Another example is the *Ambiguous names* anti-pattern: since in Java method returns do not have a name, WSDL generation tools tend to name return messages with generic names. Therefore, automatically generated WSDL documents usually have ambiguous named messages. However, if the method signatures do not have ambiguous names, the quantity of ambiguous names in the generated WSDL documents decreases. All in all, this approach does not aim at obtaining optimal WSDL documents in terms of anti-pattern occurrences, but at generating good quality WSDL documents, while preserving the advantages of the code-first methodology, namely low development cost and fast time-to-market [31, 27].

Regarding future work, we are studying how to improve the accuracy of the different anti-pattern cause detectors, such as lowering the false positive rate of the Inappropriate comments detector. For example, we are planning to enhance it using concept disambiguation. To do this, we are considering tools such as WikipediaMiner[††], to assess how terms and concepts are connected to each other based on content published in Wikipedia [43]. We are also extending the detectors for supporting other languages, such as Spanish or Portuguese. We will also extend our WSDL document generation tool to support other programming languages, such as C#. Finally, we will explore heuristics for semi-automatically refactoring Web Services source code. For instance, if a parameter name is ambiguous, our idea is to analyze the class of the parameter looking for potential alternative names. In addition, if the service implementation uses that parameter as a parameter to call another method, the name positionally associated in the latter method to that parameter might provide hints about more suitable names for the ambiguous parameter. We will also explore whether linguistic corpuses, such as WordNet *[34]* or Wikipedia *[43]*, can be used to narrow down the potential names by means reducing set of words into a synonym or a hypernym. Indeed, this is an interesting but difficult problem that is worth to explore.

Another research opportunity involves analyzing how discoverable the WSDL documents generated with AF-Java2WSDL are compared to other approaches to anti-pattern avoidance [30].

[††] http://wikipedia-miner.cms.waikato.ac.nz/

The goal of this comparison is to assess which is the benefit of attacking the various anti-pattern causes directly (as our tools does) versus the indirect approach to anti-pattern removal followed by [30].

Web Service discovery is also important in the context of Web Service mash-up [44, 45, 46], which means to combine different Web Services to obtain a larger, aggregate functionality. Different works have pointed out that discovering Web Services is essential not only to select the Web Services to be combined but also to maintain high-quality interfaces in the mash-up [44, 45]. In [46], the authors present an approach for Web Service mash-up that partially relies on Web Service discovery. Furthermore, the authors state that some operation/method naming practices, which they call naming tendencies [18], have to be considered when mashing-up Web Services to produce understandable mash-ups. In addition, their approach disregards some unrepresentative names, such as "result" or "return", because they cannot be used to describe a Web Service functionality. As discussed in Section 2, our anti-pattern definitions include naming tendencies as well as other practices that hinder Web Service discovery. Basically, the anti-patterns negatively impact on Web Services registries because they negatively affect text-based analysis of WSDL documents, such as classification and clustering. Since the approach proposed in [46] heavily relies on this kind of analysis, a future work will be to assess the impact of our work when producing Web Services that are likely to be mashed-up into larger services.

Finally, we are planning to evaluate the benefits of the practices that our tool enforces in other aspects of Web Services, such as usability or replaceability. This is important because Web Services are being used in new contexts in which discoverability is not understood as finding a Web Service, but finding who provides a particular Web Service. One example of this context are Home Services [47], where service providers are connected to a home network and service consumers should search for who provides a particular service using protocols such as WS-discovery[‡‡].

ABOUT THE AUTHORS

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

Juan Manuel Rodriguez (http://www.exa.unicen.edu.ar/~jmrodri) obtained a Ph.D. degree in Computer Science from the UNICEN, in 2012. His thesis was about quality of Web Services APIs. He was founded by the Argentinian National Council for Scientific and Technical Research (CONICET) and worked under the supervision of Alejandro Zunino. He holds a Systems Engineer degree from the UNICEN. He is a member of ISISTAN Research Institute.

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

ACKNOWLEDGMENTS

We acknowledge the financial support provided by ANPCyT through grant PICT-2012-0045.

[‡‡]WS-discovery: http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.html

REFERENCES

[1] Papazoglou M, Traverso P, Dustdar S, Leymann F. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems* 2008; **17**(2):223–255.

[2] Erl T. *SOA Principles of Service Design*. Prentice Hall, 2007.

[3] Crasso M, Rodriguez JM, Zunino A, Campo M. Revising wsdl documents: Why and how. *IEEE Internet Computing* 2010; **14**:48–56.

[4] Rodriguez JM, Crasso M, Zunino A, Campo M. Improving web service descriptions for effective service discovery. *Science of Computer Programming* 2010; **75**(11):1001 – 1021.

[5] Gurugé A. 3 - microsoft's web services. *Web Services*. Digital Press: Burlington, 2004; 95 – 143.

[6] Kastner C, Thum T, Saake G, Feigenspan J, Leich T, Wielgorz F, Apel S. Featureide: A tool framework for feature-oriented software development. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009; 611 –614.

[7] Benatallah B, Casati F, Toumani F. Web service conversation modeling: a cornerstone for e-business automation. *Internet Computing, IEEE* 2004; **8**(1):46 – 54.

[8] Gronmo R, Skogan D, Solheim I, Oldevik J. Model-driven Web Services development. *e-Technology, e-Commerce and e-Service, 2004. EEE '04. 2004 IEEE International Conference on*, 2004; 42 – 45.

[9] Vara JM, de Castro V, Marcos E. WSDL automatic generation from UML models in a MDA framework. *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, 2005; 6.

[10] Baïna K, Benatallah B, Casati F, Toumani F. Model-driven Web Service development. *Advanced Information Systems Engineering*, *Lecture Notes in Computer Science*, vol. 3084, Persson A, Stirna J (eds.). Springer Berlin Heidelberg, 2004; 290–306.

[11] Santos RL, Roberto PA, Gonçalves MA, Laender AH. A web services-based framework for building componentized digital libraries. *Journal of Systems and Software* 2008; **81**(5):809 – 822.

[12] Kim JB, Segev A. A web services-enabled marketplace architecture for negotiation process management. *Decision Support Systems* 2005; **40**(1):71 – 87.

[13] Bhagat J, Tanoh F, Nzuobontane E, Laurent T, Orlowski J, Roos M, Wolstencroft K, Aleksejevs S, Stevens R, Pettifer S, *et al.*. Biocatalogue: a universal catalogue of web services for the life sciences. *Nucleic Acids Research* 2010; **38**(suppl 2):W689–W694.

[14] Zhu D, Li Y, Shi J, Xu Y, Shen W. A service-oriented city portal framework and collaborative development platform. *Information Sciences* 2009; **179**(15):2606 – 2617.

[15] Dong X, Halevy A, Madhavan J, Nemes E, Zhang J. Similarity search for web services. *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, VLDB Endowment, 2004; 372–383. URL http://dl.acm.org/citation.cfm?id=1316689.1316723.

[16] Stroulia E, Wang Y. Structural and semantic matching for assessing Web Service similarity. *International Journal of Cooperative Information Systems* Jun 2005; **14**(4):407–438.

[17] Garofalakis J, Panagis Y, Sakkopoulos E, Tsakalidis A. Contemporary Web Service Discovery Mechanisms. *Journal of Web Engineering* 2006; **5**(3):265–290.

[18] Blake MB, Nowlan MF. Taming Web Services from the wild. *IEEE Internet Computing* 2008; **12**(5):62–69.

[19] Crasso M, Zunino A, Campo M. Combining query-by-example and query expansion for simplifying Web Service discovery. *Information Systems Frontiers* 2011; **13**(3):407–428.

[20] Al-Masri E, Mahmoud QH. WSB: a broker-centric framework for quality-driven web service discovery. *Software: Practice and Experience* 2010; **40**(10):917–941.

[21] Hao Y, Zhang Y, Cao J. Web services discovery and rank: An information retrieval approach. *Future Generation Computer Systems* 2010; **26**(8):1053 – 1062.

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

[23] Wu C. Wsdl term tokenization methods for ir-style web services discovery. *Science of Computer Programming* 2012; **77**(3):355 – 374.

[24] Basili V, Briand L, Melo W. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* Oct 1996; **22**(10):751 –761.

[25] Pasley J. Avoid XML schema wildcards for Web Service interfaces. *Internet Computing, IEEE* May-June 2006; **10**(3):72–79, doi:10.1109/MIC.2006.45.

[26] Beaton J, Jeong SY, Xie Y, Jack J, Myers BA. Usability challenges for enterprise service-oriented architecture APIs. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2008; 193–196, doi:10.1109/VLHCC.2008.4639084.

[27] Rodriguez J, Crasso M, Mateos C, Zunino A, Campo M. Bottom-up and top-down cobol system migration to web services: An experience report. *Internet Computing, IEEE* March-April 2013; **17**(2):44–51, doi:10.1109/MIC.2011.162.

[28] Rodriguez JM, Crasso M, Zunino A. An approach for web service discoverability anti-patterns detection. *Journal of Web Engineering* 2013; **12**(1–2):131–158.

[29] Rodriguez JM, Crasso M, Mateos C, Zunino A. Best practices for describing, consuming, and discovering web services: A comprehensive toolset. *Software: Practice and Experience* 2012; **In press**.

[30] Mateos C, Crasso M, Zunino A, Coscia JLO. Detecting WSDL bad practices in code-first web services. *International Journal of Web and Grid Services* 2011; **7**:357–387.

[31] Rodriguez JM, Crasso M, Mateos C, Zunino A, Campo M, Salvatierra G. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, chap. The SOA Frontier: Experiences with Three Migration Approaches. IGI Global, 2013; 126–152.

[32] Rodriguez J, Crasso M, Zunino A, Campo M. Automatically detecting opportunities for Web Service Descriptions improvement. *Software Services for e-World*, *IFIP Advances in Information and Communication Technology*, vol. 341, Cellary W, Estevez E (eds.), Springer Boston, 2010; 139–150.

[33] Dan Klein CDM. Accurate unlexicalized parsing. *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, 2003.

[34] R Feldman JS. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2006.

[35] Chae HS, Kwon YR, Bae DH. A cohesion measure for object-oriented classes. *Software: Practice and Experience* 2000; **30**(12):1405–1431.

[36] Allen EE, Cartwright R. Safe instantiation in generic java. *Science of Computer Programming* 2006; **59**(1-2):26 – 37. Special Issue on Principles and Practices of Programming in Java (PPPJ 2004).

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

[38] Heß A, Johnston E, Kushmerick N. ASSAM: A tool for semi-automatically annotating semantic Web Services. *International Semantic Web Conference*, *Lecture Notes in Computer Science (LNCS)*, vol. 3298, 2004; 320–334.

[39] Al-Masri E, Mahmoud QH. QoS-based discovery and ranking of web services. *16th International Conference on Computer Communications and Networks (ICCCN 2007)*, 2007; 529–534.

[40] Henning M. API design matters. *Communications of the ACM* May 2009; **52**(5):46–56.

[41] Salton G, AWong, Yang CS. A vector space model for automatic indexing. *Communications of the ACM* 1975; **18**(11):613–620.

[42] Agichtein E, Brill E, Dumais S, Ragno R. Learning user interaction models for predicting web search result preferences. *29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06), Seattle, Washington, USA*, ACM Press: New York, NY, USA, 2006; 3–10.

[43] Milne D, Witten IH. An open-source toolkit for mining Wikipedia. *Artificial Intelligence* 2013; **194**(0):222 – 239.

[44] Zhang MW, Zhang B, Liu Y, Na J, Zhu ZL. Web service composition based on qos rules. *Journal of Computer Science and Technology* 2010; **25**(6):1143–1156, doi:10.1007/s11390-010-9395-0. URL http://dx.doi.org/10.1007/s11390-010-9395-0.

[45] Jiang W, Wu T, Hu SL, Liu ZY. Qos-aware automatic service composition: A graph view. *Journal of Computer Science and Technology* 2011; **26**(5):837–853, doi:10.1007/s11390-011-0183-2. URL http://dx.doi.org/10.1007/s11390-011-0183-2.

[46] Blake M, Nowlan M. Knowledge discovery in services (kds): Aggregating software services to discover enterprise mashups. *Knowledge and Data Engineering, IEEE Transactions on* 2011; **23**(6):889–901, doi:10.1109/TKDE.2010.168.

[47] Wei M, Xu J, Yun H, Xu L. An open-source toolkit for mining Wikipedia. *Computer Science and Information Systems* 2012; **9**(2):813 – 838.