Journal of Web Engineering, Vol. 14, No. 1&2 (2015) 251–285

ASSISTING DEVELOPERS TO BUILD HIGH-QUALITY CODE-FIRST WEB SERVICE APIS

JUAN MANUEL RORIGUEZ CRISTIAN MATEOS ALEJANDRO ZUNINO
ISISTAN-CONICET, UNICEN, Paraje Arroyo Seco S/N
Tandil, Buenos Aires, Argentina
{*juanmanuel.rodriguez, cristian.mateos, alejandro.zunino*}*@isistan.unicen.edu.ar*

Received July 14, 2014
Revised January 2, 2015

Searching and understanding Web Services is challenging mainly because service developers tend to disregard the importance of the exposed service APIs. In many situations these APIs are defined using WSDL documents, which are written in an XML dialect. Service developers can opt between manually writing these documents or inferring them from service implementations. This work is focused on the later methodology and provides techniques and tools for generating WSDL documents avoiding well-known bad practices (e.g., lack of textual comments or representative names) that compromise Web Service API quality. Our experiments show that by using our approach the obtained WSDL documents are more likely to be free from these undesirable practices so they better describe service functionality. In addition, these experiments show that the WSDL documents generated using our approach also have other important quality-related features, such as less redundant data-type definitions and better defined data-types. Experiments have been performed by comparing our proposal against WSDL generation tools/approaches from both the industry and the academy.

Keywords: WEB SERVICES, CODE-FIRST, WSDL ANTI-PATTERNS, AUTOMATIC DETECTION, SERVICE DISCOVERY, SERVICE UNDERSTANDABILITY
Communicated by: D. Schwabe & O. Diaz

1 Introduction

Web Service is the commonest technology for implementing the Service-Oriented Computing (SOC) paradigm [30, 13]. 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 sets of atomic operations that can be remotely invoked using the aforementioned Web protocols. Web Service operations and technical details can be described using the Web Service Description Language (WSDL). Since all the information required for invoking a service is contained in its WSDL document, service consumers do not need to know service implementation details. In addition, WSDL documents are useful for indexing services in registries so service consumers can look for the services they need [10, 43].

Although WSDL documents are intended to help service consumers, publicly available WSDL documents are in general difficult to read and tend to have neither proper comments nor descriptive names in their elements [15, 4, 3, 8]. We refer to this kind of WSDL documents as low-quality WSDL documents. As a result of this, service consumers have the difficult task of finding the Web Services they need based on low-quality WSDL documents. This is because many Web Service registries index the information present in WSDL documents [10]. This might not seem relevant because there are techniques for discovering Web Services through semantic-based discovery approaches, such as those

exploiting DAML-S[a]or OWL-S[b]service meta-data [16, 28]. However, these approaches are difficult to be applied because of the complexity of defining the necessary ontologies and describing the Web Services using these ontologies [10, 43, 9]. Therefore, there are currently many domains in which searching Web Services through their WSDL documents is the only viable option [25, 43].

An example illustrating WSDL document importance is presented in [43] where WSDL documents are used to search and mash-up Web Services. In their motivation example, the authors present six services that are only described by means of WSDL documents. The problem is to obtain the weather forecast for an area given by a MSISDN (one kind of phone number) and the area diameter. However, in the set of services there is not a single Web Service to do this. Therefore, their approach combines the Web Services, using their descriptions, to solve the problem. Given the MSISDN and the area, their approach uses a Web Service to obtain the name of the city and the district code. Then, using this information, the approach selects another service to get the city latitude and longitude. Finally, the approach invokes a Web Service that given a latitude and longitude returns the weather forecast for that area. Notice that there is no other constraint to do so but describing the Web Services using the standard WSDL document. Therefore, this approach –as well as many others providing service composition mechanisms– can be applied to any set of standard Web Services. In their example, the authors assume that the WSDL documents contain proper names that allow users to understand Web Services input and output parameters.

Another drawback of low quality WSDL documents is that service consumers have problems to understand them [8]. This is important even when there is no discovery involved because a WSDL document is in essence an application-programming interface (API) definition. Several researchers [5, 20, 42, 18, 31] have pointed out that certain issues in names, comments and structure can render APIs difficult to be used and understood. Therefore, using low quality APIs results in hidden costs [20]. There are real situations, such as the one presented in [33], that evidence the failure of a SOC system due to low quality WSDL documents. Basically, [33] discusses an enterprise COBOL system migration to SOC architecture. During a first migration attempt, developers disregarded WSDL document quality factors. As a result, the migrated system Web Services were complex to be used by third-parties. This complexity originated elevated costs when developing software that consume these Web Services. This fact was one of the main issues that led the system owner to remigrate the system, but this time WSDL document quality was a main requirement. The study shows that WSDL document quality is an issue not only in publicly available Web Services, but also in nowadays enterprise Web Services, even when discovery is not part of consumer application life-cycle.

Although several works [3, 15, 34] have also pointed out that WSDL document quality is an important aspect usually disregarded by Web Service developers, there is little work [32, 35, 29] aimed at helping or assisting Web Service developers to produce good quality WSDL documents. Therefore, this work proposes an approach for assisting developers to improve WSDL document quality in Java-based, code-first Web Service development. Code-first means to implement the Web Service logic first and automatically derive the WSDL documents that describe the services later. Therefore, our approach consists in detecting issues in the service code that the current WSDL generation tools might otherwise map to problematic WSDL documents. Some previous works in this line [32, 35] only aim at spotting low-quality WSDL documents produced by means of the not-so-popular contract-first

[a] DAML-Shttp://www.daml.org/services/
[b] OWL-Shttp://www.w3.org/Submission/OWL-S/

Web Service development. In contrast, the authors of [29] propose early code refactorings to improve generated WSDL document quality, but the approach does not cover all the WSDL document *anti-patterns* or bad practices [34]. For instance, this approach does not propose any refactoring to improve comments' quality.

In addition to the techniques for detecting code issues that might lead to anti-patterns in automatically generated WSDL documents, this work briefly presents an Eclipse plug-in that implements the techniques. As a result, these techniques can be easily integrated into real-life code-first Web Service development projects without much effort. The plug-in analyzes a Web Service source code, detects possible issues and suggests suitable refactorings to remove them. The plug-in also provides a custom WSDL generation tool that avoids other problems that are inherent to WSDL generation rather than service code programming. The plug-in is open source, available upon request, and the binary version can be freely downloaded from its homepage[c].

All in all, previous works [34, 8] have pointed out that several common practices in WSDL documents, called anti-patterns, affect Web Service discoverability. In addition, a real-life case study [33] has shown that these anti-patterns can lead large Service Oriented system development to failure. To assist developers, we have developed techniques [35] for automatically detecting anti-patterns in WSDL documents, so developers can refactor these documents. However, developers commonly do not write WSDL documents manually because they are automatically generated from the Web Service implementation. Although there is an study [29] that aims at reducing the impact of these anti-patterns on code-first Web Services, the study does not focus on the anti-pattern causes, but on the correlation between their occurrences and well-known Object-Oriented (OO) metrics. In this context, this work presents:

- novel techniques for detecting anti-pattern causes in Web Service implementations, some of which are adaptations of the techniques previously proposed for detecting anti-patterns in WSDL documents [35].
- a WSDL document generation tool that takes into account the anti-patterns presented in [34].
- an empirical evaluation of the effectiveness of both the detection techniques and the generation tool w.r.t. anti-pattern avoidance.
- a comparison against the OO metric based approach in [29], which as far as we know is the only approach aiming at preventing the anti-patterns in code-first Web Services.
- an Eclipse plug-in for simply adopting both the detection techniques and the generation tool in real Java-based Web Service development.

The rest of the paper is organized as follows. Firstly, Section 2 discusses related work for improving WSDL document quality. Secondly, Section 3 presents our approach for improving the quality of WSDL documents resulting from code-first Web Service development. Then, Section 4 presents the experimental evaluation, comparing our approach against the only related proposal developed so far. Section 5 describes our Eclipse plug-in. Section 6 outlines this paper findings and the approach limitations. Finally, Section 7 discusses future works that might overcome the current limitations of this approach.

[c] https://sites.google.com/site/easysoc/home/code-first-assistant

2 Background and Related Works

Broadly, evaluating software quality has been always a major concern for developers [2]. Therefore, different methodologies to assess a wide range of quality aspects in software artifacts, such as readability, functional cohesion or complexity, have been developed. Particularly, several researchers [5, 3, 20, 42, 18, 31] have pointed out the importance of good quality API. APIs are public component definitions of a software that allow third-party software to interact the former. For example, the OpenGL API allows software to render 3D graphics independently of the underlying technology. The type and semantics of the components vary according to the technology used, e.g., functions in C, objects and methods in Java, and port-types and operations in WSDL. Despite their differences, different authors showed that there are common problems in API designs [20, 42, 31]. Since APIs are intended to be used by developers [3, 18], many of the reported issues are related to how easily developers understand what an API does. Mainly, API problems are related to comments, names, data-type structures and API components aggregation by semantic cohesion. Although these problems have subjective factors that make it difficult to automatically assess API quality, there are works that aim at quantifying some of these factors [12, 24, 21, 31].

Despite the importance of comments and names, in [18], the authors have reported that up to 35% of the issues found in a particular API are in the comments, i.e., the components have low quality comments, if they have any at all. Furthermore, more than 12% of this API issues are related to poor naming. Even more, these naming issues do not include another 12% of the problems that are categorized as consistency and conventions issues. Other issues detected in the API are related with data-type definition and error handling. Another study [42] that analyzed a larger set of APIs have pointed out that more than 30% of the APIs elements have comment-related issues. Moreover, 10.3% of the APIs elements are over or under exposed. In addition, 1.9% and 4.2% of the API elements are affected by naming and consistency issues.

In both [12] and [24], the authors present methods to assess name quality during software development. Basically, these works propose to map names to a set of manually defined concepts that are relevant in the software domain. In addition, developers have to define a partial order relationship among these concepts. Having this, desirable properties can be easily defined and verified. For instance, a desirable property can be "there is only one name associated to each concept" or "the name of a specific concept includes the name of the most generic concept". The main drawback of these approaches is that it is necessary to manually create and update the concept set, define the order among then and map the names to concepts. Therefore, these approaches might be suitable for new developments, but it is difficult to apply these approaches in an ongoing software development [21].

Contrary to the approaches presented in [12, 24], the approach discussed in [21] is able to automatically analyze method names, which are a subset of the identifiers in software code. Particularly, this approach targets software developed in Java. The approach analyzes the method names composition using a part-of-speech tagger based on WordNet [26] and an ad-hoc dictionary. The tagger basically maps the name of a method to its structure. For instance, the name `getSize()` can be mapped to <verb>-<noun>. On the other hand, the approach automatically assesses the methods to determine whether they have certain properties, which are called method attributes. [21] defines a set of attributes that can be easily spotted in a method. Some of the attributes are the method return type, if the type is referred to in the method name, if the method reads an instance variable, and if there is a loop in the method. Using names, name tags and method attributes, the authors proposed to derive rules from a representative Java project corpus. The rules have the following form: "In the 99% of the cases that

the name has the form `contains*` , where * is a wild-card, the method returns a boolean". According to the authors, finding a method with that name structure that returns another data-type is equivalent to finding a bad name. They state that since the method has a strange behavior, it would be difficult for developers to understand what the method does.

Regarding comments, it is well-known that they are essential in software development to understand a piece of code [39]. Furthermore, the industry has acknowledged that comments are an acceptable and useful information source to document APIs. For example, Javadoc[d] and Doxygen[e] are two widely used tools that extract comments from source code to create HTML pages that document the API. In this context, there are different works [23, 37] aiming at assessing comment quality. According to [21], the better the source code comments are, the fewer bugs the source code has. Both works [23, 37] assess comment quality from the stand point of a developer that has access to the code. Although these works do not assess the quality of comments as API documentation, they show that it is possible to automatically or semi-automatically assess comments quality.

Concerning Web Services, researchers have studied them from different angles, such as interoperability, security, or interface quality. For instance, the Web Services Interoperability Organization (WS-I), an organization that establishes best practices for interoperability, has released several profiles.[f] A profile describes how to implement interoperable services based on the exhaustive analysis of popular enterprise software middlewares, namely .NET and JEE. In addition, the WS-I provides testing tools, e.g., BP 1.2 and 2.0 Interoperability Test Suites, to ensure that a Web Service implementation satisfies these profiles. Furthermore, other works aim to improve Web Services performance [14, 38] and reliability [36].

Other works [15, 4] have studied text related code issues that jeopardize WSDL understandability and discoverability. In [15], the authors analyzed the comments present in a real-life WSDL document data-set. To analyze those WSDL documents comments, the authors crawled public registries on the Internet. After that, the authors gathered 640 WSDL documents and analyzed the lengths of the textual comments of these services (including the registration information and all the comments present in the WSDL documents). The results show that in the 80% of the documents the average documentation length per operations is 10 words or less. Furthermore, from this 80%, half of them have no documentation at all. In addition, 80% of the comments have less than 50 words, while 52% of the comments have less than 20 words. Moreover, in [4], the authors analyze elements names in WSDL documents. The authors detect "naming tendencies" in the names of WSDL document elements and empirically show that these tendencies negatively impact the retrieval effectiveness of a syntactic registry. They analyzed the names of message parts that belong to 596 WSDL documents gathered from Internet repositories. The name of each message part was compared against the remaining part names, and then four naming tendencies were observed in service message parts. Broadly, these tendencies show that developers use common phrases within part names. For example, a message part standing for a user's name is commonly called "name", "lname", "user_name" or "first_name" [4].

In particular, [34] presents an anti-pattern catalog for assessing *understandability* and *discoverability* in WSDL documents, which are aspects related to interface quality. The catalog defines eight common issues that are known to negatively affect Web Service understandability and discoverability. These issues are presented in the form of anti-patterns, which are briefly depicted in Table 1. These

[d] Javadoc: `http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html`
[e] Doxygen: `www.doxygen.org/`
[f] Basic Profiles: `http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile`

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
Ambiguous names	Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent
Low cohesive operations in the same port-type	Occurs when port-types have weak functional cohesion.	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

Table 1. Catalog of Web Service anti-patterns

anti-patterns are related to issues in data-type structure, service functional aggregation, names and comments within Web Service WSDL documents. Therefore, the anti-patterns describe issues that are similar to the API issues that affect API usability or understandability [5, 20, 42].

As API issues might generate great cost in terms of development using them [20], the anti-patterns reported in [34] might render services less usable. The case study presented in [33] shows the downside of disregarding the anti-pattern catalog in favor of speeding-up development. In this case study, a large governmental agency migrated their legacy system to SOC technology disregarding WSDL document quality. The system was rapidly developed, but developers found the resulting Web Services difficult to be used. This resulted in the necessity of performing a second migration because of high development costs due to using the original Web Services. Mainly, it was practically impossible for developers that were not involved in the main system development to use the obtained Web Services. This case study presented evidence that the anti-patterns not only affect Web Service discoverability, but also Web Service understandability.

In [32], the authors propose algorithms and heuristics for automatically detecting some of the anti-patterns presented in [34], or "anti-pattern detectors", designed for analyzing WSDL documents. The work was extended in [35] to improve the performance of the previously defined heuristics. Since the detectors presented in [32] and [35] take as input WSDL documents, these detectors are suitable for assisting contract-first Web Service development. This methodology proposes to write the WSDL document first and then implement the associated Web Service logic. However, code-first Web Service development is the usual way of implementing Web Services in the industry, which means that developers do not manually modify the WSDL documents [29]. Yet, these detectors might be useful to assess automatically generated WSDL documents from the point of view of the heuristics they rely on. However, the potential anti-pattern causes in code-first WSDL documents might come from two different sources, namely the code implementing the associated service and the WSDL generation tool used. Thus in code-first development an anti-pattern prevention approach, rather a detection tool such as [32, 35], is needed.

Regarding code-first Web Service development, a recent study [29] has pointed out that there are statistical correlations between some classical OO source code quality metrics and the presence of anti-patterns in generated WSDL documents. In particular, this study has focused on the well-known metrics defined in the Chidamber and Kemerer's catalog [7], such as Abstract Type Count and Coupling Between Objects. Therefore, applying refactorings to improve these metrics might result in better quality in WSDL documents. These might not be always the case because some of the relevant correlations are as low as 0.6 ($p\text{-}value < 0.05$). Therefore, this indirect approach to anti-pattern prevention cannot effectively cover all anti-pattern causes. 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, or adding/removing parameters. However, these refactorings do not cover all the code issues that might led to anti-patterns in generated WSDL documents. For example, this approach does not consider are absence or inappropriate comments, and meaningless or ambiguous parameter and method names.

This paper analyzes the possibility of adapting some of the techniques presented in [32] as well as presenting new ones for detecting code issues in Web Service source codes that could be mapped into structural and textual anti-patterns in generated WSDL documents. We complement this with a new WSDL generation tool capable of building WSDL documents by also avoiding the anti-patterns that are inherent to WSDL generation rather than service implementation practices.

3 Assisting code-first Web Service developers

Basically, code-first Web Service development consists in writing the logic of a Web Service in a particular language and then using tools for generating the WSDL documents and exposing this logic as a Web Service. A previous work [29] has shown that the way the Web Service logic is implemented can affect the quality of the resulting Web Service in terms of discoverability and understandability. Moreover, in code-first Web Services, the anti-patterns present in WSDL documents might stem from two non mutually exclusive sources: their associated service implementations or the WSDL generation tool used. For instance, if there are no comments in the service implementation, the generation tool has no comments to place in the WSDL document. However, defining a port-type more than once is a result of how the WSDL document is generated. Therefore, our approach consists in assisting code-first Web Service developers to write good Web Service code and then to correctly map it into high quality WSDL documents.

For designing the approach, we analyzed the discoverability anti-patterns from the literature and which can be the possible causes of them in code-first WSDL documents. Table 2 summarizes this analysis. Some of the anti-patterns have only one cause. For instance, *Enclosed data model* anti-pattern results only from the WSDL generation tool because it decides where to place data-type definitions independently of how classes are defined in the Java code. In contrast, other anti-patterns, or sub anti-patterns, might stem from either source code or the WSDL document generation tool. This is the case for *Lacking comments* anti-pattern, which might stem from either lacking comments in the source code or the WSDL document generation tool ignoring the comments.

Taking into account that there are two sources for anti-patterns when implementing Web Services through code-first, our approach is divided into two parts. The first part of our approach is focused on the anti-patterns that might arise from poor Web Service implementation practices. As stated in Table 2, these anti-patterns are *Inappropriate or lacking comments* , *Ambiguous names* , *Low cohesive operations in the same port-type* , *Undercover fault information within standard messages* and *Whatever types* . Basically, our approach consists in analyzing the classes that will be exposed as services looking for possible issues that could translate into anti-patterns. To perform this analysis, a detection heuristic, *detector* for brevity sake, has been developed for each anti-pattern.

Using the results of the classes' analysis, the developer should manually refactor them. Table 3 presents the refactorings for each potential issue. The refactorings are based on the ones originally proposed in [34] for the WSDL documents. However, they are adapted to Java specifics, such as commenting using JavaDoc tags or correctly using generics. To further illustrate how the original refactorings were adapted, consider for instance the *Undercover fault information within standard messages* anti-pattern. The original refactoring for this anti-pattern is to remove error information from output messages and define fault messages. However, Java does not handle errors in such way, but defines Exceptions. Therefore, the Java refactoring for this anti-pattern cause is to remove error information from the classes associated to method results and raise a Java Exception[g] instead. Although correctly applying the refactorings heavily depends on the developer's skills and expertise, once they are applied, the developer can reanalyze the classes to verify that they now are anti-pattern cause free.

The approach second part is a tool for mapping the classes that would be exposed as Web Services into good quality WSDL documents. This tool considers all the names and comments that the developer placed in the source code of the service implementation to obtain well-named and well--

[g] The Java® Language Specification. Java SE 8 Edition. Chapter 11: Exceptions: `http://docs.oracle.com/javase/specs/jls/se8/html/jls-11.html`

Anti-pattern	Sub anti-patterns	Source code cause	WSDL generation tool cause
Enclosed data model	*Enclosed data model*	-	Embedding the XSD schema for complex data-types within the WSDL document.
Redundant port-types	*Redundant port-types*	-	Replicating the mapped interface for each supported transport protocol.
Redundant data models	*Redundant data models*	-	Replicating a data-type definition each time it is used.
Inappropriate or lacking comments	*Lacking comments*	Not commenting methods in the class that offers the service.	Ignoring the comments in the original source code.
	Inappropriate comments	Commenting methods with irrelevant information.	-
Ambiguous names	*Ambiguous names in operations*	Using unrepresentative names for methods.	-
	Ambiguous names in parameters	Using unrepresentative names for parameters.	Ignoring parameters names and using unrepresentative names for elements that are anonymous in the source code but must be named in the WSDL document.
Low cohesive operations in the same port-type	*Low cohesive operations in the same port-type*	Adding uncohesive methods in the class that will be exposed as a service.	-
Undercover fault information within standard messages	*Undercover fault information within standard messages*	Using control couples instead of exception mechanism.	Mapping exceptions into the data-types used by output messages instead of using WSDL fault messages.
Whatever types	*Whatever types*	Using generic data-type/classes, such as the Object class.	Replacing complex data-types for the xsd:any type.

Table 2. Anti-patterns characterization and causes

Anti-pattern	**Sub anti-pattern**	**Refactoring in Java Source Code**
Inappropriate or lacking comments	*Lacking comments*	Adding comments in JavaDoc form to uncommented methods.
	Inappropriate comments	Rewriting comments to convey information about the method semantics.
Ambiguous names	*Ambiguous names in operations*	Rename operations using relevant terms. Verify that the name has verb+noun syntax.
	Ambiguous names in parameters	Rename operations using relevant terms. Verify that names are nouns or noun phrases.
Low cohesive operations in the same port-type	*Low cohesive operations in the same port-type*	Cluster methods according to their semantic following the detector recommendation and separate them in different facade classes.
Undercover fault information within standard messages	*Undercover fault information within standard messages*	Remove control tuples and raise errors as Java Exceptions.
Whatever types	*Whatever types*	1. Replace all parameter and return types defined as Object by a more concise class. 2. Parametrize Java generics with concise classes instead of using Object or the wild-card “?”

Table 3. Anti-pattern cause refactoring

commented WSDL documents. In addition, the tool was designed to avoid WSDL generation issues such as replicating port-types. Finally, the tool also attempts to minimize the number of repeated types and correctly using the fault mechanism provided by WSDL for Web Services. Basically, both parts of our approach are designed to assist code-first Web Service developers to obtain better quality WSDL documents, in discoverability and understandability terms.

Section 4.1 presents the detection approach, while Section 4.2 describes the WSDL document generation approach. To clarify the proposed techniques, both sections will consider an example Web Service Java source code. This example is a fictitious Web Service which offers access to the catalog of a fictitious on-line CD/Book store. The following code is part of the Java source code for the example Web Service:

```
//Output Class
public class Book {
        private String title;
        private String ISBN;
        private String author;
        private int pages;
        ....
        //Getters and Setters
        ....
}
//Output Class
public class CD {
        private boolean nonExistingCDError;
        private String artist;
        private String album;
        private int numTracks;
        private Track[] tracks;
        ....
        //Getters and Setters
        ....
}
//Service Implementation
public class BookStoreCatalog {
        /** Service for costumers.*/
        public String getBookTitle(String ISBN){...}
        public String getBookAuthor(String ISBN) {...}
        /** Returns a book given its ISBN.*/
        public Book getBook(String ISBN) {...}
        /**Retrieves a CD information using the store SKU.*/
        public CD getCD(int sku) {...}
}
```

3.1 Automatically detecting discoverability issues

This section presents the different heuristics for detecting issues in the Java implementation of Web Services that can result in anti-patterns in the generated WSDL documents. All the detectors materializing these heuristics are designed to detect issues in the front-end or API class that would be

exposed as a service, i.e., the class that would be used as input by the WSDL document generation tool. The current section is divided into five subsections, one per anti-pattern cause detection heuristic. Subsection 3.1.1 presents the heuristics for *Inappropriate or lacking comments* causes detection. In Subsection 3.1.2, the heuristics for detecting *Ambiguous names* causes are discussed. Subsection 3.1.3 describes the method for detecting *Low cohesive operations in the same port-type* causes. Then, Subsection 3.1.4 introduces the heuristic for *Whatever types* causes detection. Subsection 3.1.5 outlines the heuristic for *Undercover fault information within standard messages* causes detection. Finally, Subsection 3.1.6 presents a brief discussion about the presented detectors and their potential limitations.

3.1.1 Inappropriate or lacking comments *causes detection*

For detecting the *Inappropriate or lacking comments* anti-pattern causes, the detector verifies that all public methods in the class that would be exposed as a Web Service have a comment, and when some method lacks comments a problem is informed to the developer. This aims at preventing the *Lacking comments* sub anti-pattern. An example of a method affected by the *Lacking comments* sub anti-pattern causes would be `getBookAuthor` . Then, the detector analyzes the existing comments and compares them with the method name and method parameter names for detecting *Inappropriate comments* causes.

For comparing the comments with the names, our heuristic creates a phrase from the method name and joins it 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, considering the method `getBookTitle` that receives `ISBN` as parameter, the resulting phrase is "get book title **with** ISBN". Then, this sentence is compared with the first sentence of the comments. Notice that camel casing is the de-facto naming convention in Java, and thus this assumption is valid. For comparing these two sentences, the detector generates a tree that grammatically represents each sentence. These trees represent the meaning of the sentence by considering the words of the sentences, and their relationship with more generic words, i.e., the hypernyms[h] of these words. The first step of the tree building process is to obtain the hypernyms for each word from WordNet [26]. Notice that a particular word can have different set of hypernyms because a word can have several meanings. In contrast, some words –such as proper nouns or domain specific jargon– might have no hypernyms because they are not defined in WordNet, in which case the tree building algorithm ignores them. The detector uses these hypernyms to create the tree that represents the phrase. The second step of the tree building process is creating a tree with an artificial root and iteratively adding the hypernyms to this tree. An artificial root is used because there is no single "most generic" term in WordNet.

Algorithm 1 shows the detailed steps for adding hypernyms to the tree. To illustrate how the tree is created, we will use again the method `getBookTitle(String ISBN)` . Firstly, the detector splits the name into three words, namely "get", "book" "title", and adds "ISBN" (parameter name), using upper case characters as word beginning markers. The detector looks into WordNet for the hypernyms of the words "get", "book", "title" and "ISBN". Since ISBN is not defined in WordNet, the heuristic ignores this word. For the sake of brevity, in the example we only have considered two hypernym lists for "book" and one for "title". These hypernym lists are presented below. From left to right, each list

[h] A hypernym is a word that is more general in meaning than a given word. WordNet provides a hypernyms hierarchy. For instance, "car" is a "self-propelled vehicle", which is a "wheeled vehicle", and so on until arrive to a generic term, in this case "entity".

Algorithm 1 Hypernym addition to a tree

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

goes from the most generic concept to the most specific one:

- Book:
  - [entity→physical entity→object, physical object→whole, unit→artifact, artifact→creation→product, production→book]
  - [entity→abstraction, abstract entity→communication→written communication, written language, black and white→writing, written material, piece of writing→dramatic composition, dramatic work→script, book, playscript]
- Title:
  - [entity→abstraction, abstract entity→communication→written communication, written language, black and white→writing, written material, piece of writing→matter→text, textual matter→line→heading, header, head→title, statute title, rubric]

When building the tree, the same concepts at the same depth are merged into a single node. For example, the resulting tree only has one "entity" node with two children: "physical entity" and "abstraction, abstract entity". Notice that the commas are part of the synset name in WordNet, i.e., "abstraction, abstract entity" is treated as an atomic element.

Figure 1 depicts the hypernym tree obtained when considering the hypernym lists described above, i.e., it is a partial tree for the words "book", "title" and "get". Notice that the example tree presented in Figure 1 only considers two out of fifteen possible meanings of "book" and one out of twelve possible meanings of "title". Furthermore, each meaning can have several hypernym sets because hypernyms also can have several meanings. Also, the example does not consider the meanings of "get". Therefore, the resulting tree would have much more nodes that the example tree.

Then, the similitude between two trees is calculated as follows:

$$sim(tree_1, tree_2) = \frac{nodes(sharedSubTree(tree_1, tree_2))}{max(nodes(tree_1), nodes(tree_2))}$$

where *sharedSubTree* is a function that returns the largest tree contained in two trees and *nodes* is the amount of nodes in a tree. According to previous works, a low similitude is enough to assure that

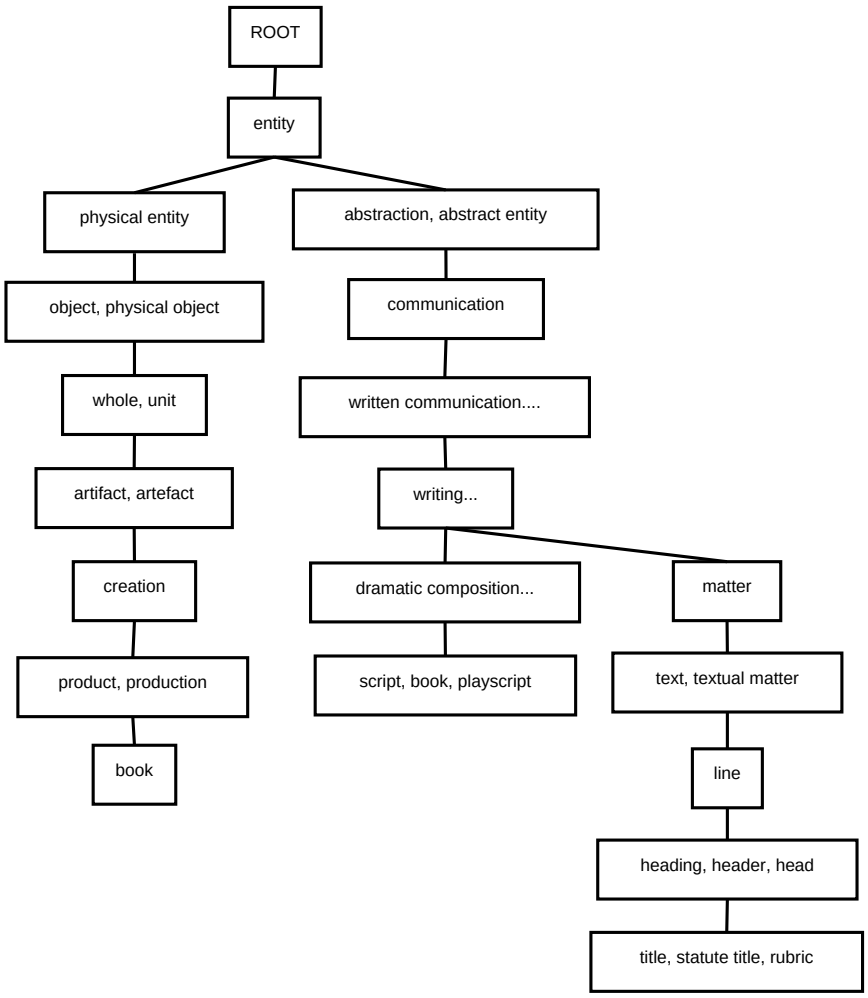

Fig. 1. Hypernym tree

comments and operation names are related [35] because the two trees –i.e., the ones obtained from the name and the comment of a method– usually have a very dissimilar number of nodes. Furthermore, the similitude tends rapidly to 0 when the sentence topics are unrelated. Therefore, a similitude higher than 0.1 is enough to assure that the comments and its operation are related [35]. The method used as example has the *Inappropriate comments* anti-pattern cause, and the described technique detects it.

3.1.2 Ambiguous names *causes detection*

For detecting *Ambiguous names* anti-pattern causes, the detector follows a three-step heuristic. The first two steps are used for detecting the causes of both *Ambiguous names in operations* and *Ambiguous names in parameters* anti-patterns. The first step verifies whether the names have an appropriate length, which should fall between 3 and 30 characters [34]. Then, the second step is to verify that the names do not use well-known unrepresentative names, namely *param, arg, var, obj, object, foo, input, output, in#, out# and str#*, where # represents a number or nothing.

Then, the third step varies for methods and parameters names. For methods names associated to the *Ambiguous names in operations* anti-pattern, the string "it must" is added at the beginning of the name to analyze. Then, the heuristic analyzes the complete string using a PCFG parser to determine whether the string has a verb and a non-pronoun, or noun, after the word "must". This is because both method and operation names should have the form "verb+noun". In contrast, parameter names are analyzed as they appear in the code. The heuristic uses the PCFG parser to determine whether parameter names are noun or noun-phrases, which is the expected form of parameter names because they represent data rather than actions. If a parameter name has any other form, it is very likely they would cause the *Ambiguous names in parameters* anti-pattern in the generated WSDL document.

The reference example is not affected by these anti-pattern causes. Firstly, all the names have a length between 3 and 30 characters. Secondly, no name is any of the unrepresentative names. Finally, all the operations names have the form `getSomething` resulting in that all operation names have the form "verb (get) + noun (book title, book author, CD)". In contrast, all the parameter names are nouns, such as ISBN, or SKU, which stands for Stock-Keeping Unit.

3.1.3 Low cohesive operations in the same port-type *causes detection*

To detect the *Low cohesive operations in the same port-type* anti-pattern causes, the tool creates an undirected graph for representing the class to be exposed as a Web Service. This graph depicts the relations among the class methods, and their parameters' names and data-types [6]. The graph has three kinds of nodes:

- Methods: this kind of nodes represents the methods of the class to be analyzed.
- Types: these nodes are the types used by the methods and all referenced types. Since primitive types and some classes, such as `java.lang.Object` or `java.lang.String` in Java, are too generic, they are complemented with the variable name, i.e., there might be two or more nodes for these particular types. 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 nodes are the singular form of the nouns, which are detected applying the PCFG parser on the method names.

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

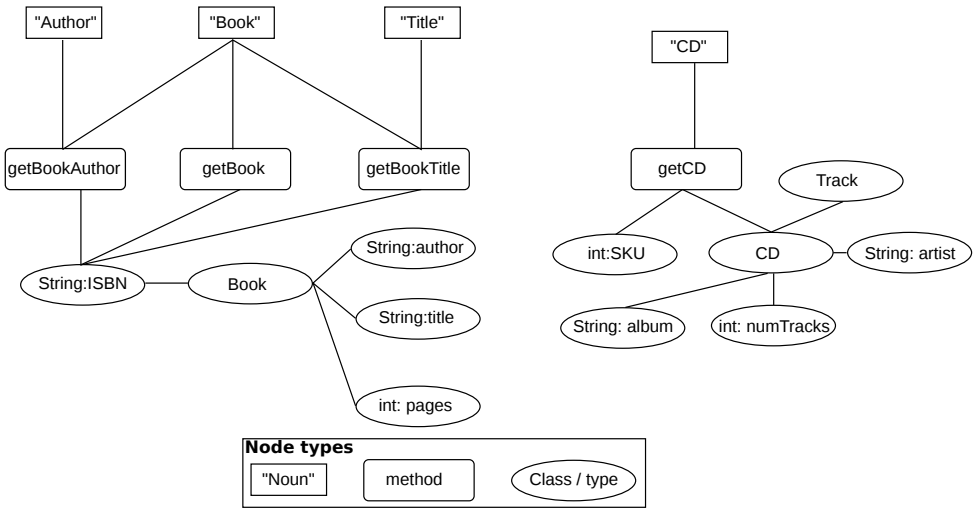

Fig. 2. Methods relationship graph

- 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 nouns: this relationship exists between a method and all the nouns in its name.

Once the graph including the methods to be exposed as a service has been generated, the detector verifies whether the graph is a connected graph. Otherwise, the detector selects the largest connected graph and reports the methods that are not represented in that graph as uncohesive. For instance, Figure 2 depicts part of the graph generated for the example. Although not all the CD class elements are represented in the figure, the resulting graph is an unconnected graph, so the detector indicates that *Low cohesive operations in the same port-type* anti-pattern causes are present. Furthermore, it would recommend moving the method `getCD` to another class, i.e., another Web Service.

3.1.4 Whatever types *causes detection*

To detect possible *Whatever types* anti-pattern causes, the detector uses a list of classes that are known to be too generic [1]. When a method uses one of these classes, the detector informs that there is an issue. These too generic classes include `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.*` . The example Web Service implementation is not affected by these anti-pattern causes. However, if there were a method defined as:

```
public List getBooks(){...}
```

Then, the source code would be affected by *Whatever types* anti-pattern causes because that `List` might contain instances of any class. In this case, one possible refactoring would be changing that `List` by another parametrized list, e.g., `List <Book >`.

3.1.5 Undercover fault information within standard messages *causes detection*

Finally, detecting the *Undercover fault information within standard messages* anti-pattern causes consist in analyzing the structure of a method output. The detector verifies whether a method has exceptions defined. If this is the case, it is likely that the method handles errors correctly. Otherwise, the detector analyzes 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 a method return value.

For instance, the method `getCD` in our example Web Service is affected by *Undercover fault information within standard messages* anti-pattern causes because the CD class has a boolean that indicates whether the CD was found. In this case, a suitable refactoring would be to remove such attribute from the CD class. Then, the developer should create a new class, namely `NoSuchCDException` , which extends of the class `Exception` and modify the method signature adding that this method might throw this new exception. Finally, it would be necessary to change the method logic to throw this exception when there is no CD with the input SKU number in the database.

3.1.6 General considerations

This part of the approach cannot tackle all the anti-patterns. Then, some of the anti-patterns cannot be avoided only by using this discussed heuristics and applying refactorings. Furthermore, even when following the enforced good practices, such as commenting the source code, there is no warranty that the generated WSDL document will not be affected by the anti-pattern related to these practices. For example, if the WSDL document generation tool disregards the comments in the source code, the generated WSDL documents would be affected by the *Lacking comments* sub anti-pattern regardless the class was commented or not. Then, it is necessary to use a tool that considers the WSDL document specification good practices, described in the following section, to complement the proposed approach.

3.2 Generating WSDL documents from service code

Our generation tool receives as input a class that implements a Web Service front-end and returns a WSDL document that describes this class and an XSD file that defines the associated data-types. Basically, the Web Service implementation API can be a Java Interface for which there is one or more classes that define the API behavior, or a Java Class that defines both the API 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 from the WSDL specification), but it can be easily extended to generate WSDL documents ready to be deployed.

The tool creates a WSDL document that exposes the public methods of a Java Interface/Class, or for brevity sake class from now on, as operations in a Web Service. For each public method of the class, the generation tool creates an operation. Each operation has the comments and name associated to the method that it maps. 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.

As a result, the tool only generates one port-type per Java class, avoiding the *Redundant port-types* anti-pattern.

For each operation, the generation tool defines an input and an output message. In addition, the tool might generate as many fault messages as exceptions that might be thrown by executing the method. Our tool 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` because it is the mechanism provided by Java to raise a fault. In both cases, the generation tool creates the XSD file by generating complex types that represent the classes, and adding recursively the required types to express all the necessary classes. The recursion ends when finding a primitive type, such as int, double or float, or a class which maps to an XSD type already defined in the XSD language, like `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 field they map. Since only one data-type is generated per POJOs, the *Redundant data models* anti-pattern is avoided as long as the POJOs are not redundant.

To illustrate how the generation tool works, we present a part of the WSDL document generated for the example code introduced at the beginning of Section 3.1. The generated code was trimmed for simplicity sake. Since the *Enclosed data model* anti-pattern impact depends on whether types 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.

```
<types>
  ...
 <xsd:element name="getBookRequest">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="ISBN" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="getBookResponse">
  ...
 </xsd:element>
 <xsd:complexType name="Book">
  <xsd:sequence>
   <xsd:element name="author" type="xsd:string" />
    ...
  </xsd:sequence>
 </xsd:complexType>
 ...
</types>
...
<message name="getBookRequest">
 <part name="parameters" element="xsd1:getBookRequest" />
</message> <message name="getBookResponse">
 <part name="parameters" element="xsd1:getBookResponse" />
</message>
...
<portType name="BookStore">
 ...
 <operation name="getBook">
  <documentation>Returns a book given its ISBN.</documentation>
  <input message="tns:getBookRequest" />
  <output message="tns:getBookResponse" />
```

```
  </operation>
...
</portType>
```

Notice that all the names in the class as well as the comments have been preserved. Since the WSDL specification defines elements that cannot be directly matched to Java elements, such as "message", some unrepresentative names were added. This is a drawback of our WSDL document generation tool. Yet, our tool as well as several other similar tools, such as Axis Java2WSDL, tries to preserve as much textual information as possible, which benefits Web Service discovery and understandability [34].

4 Experiments

To evaluate the approach, we used a data-set of 60 services written in Java. This data-set is a sub-set of services that could be fully compiled and built from those included in the service data-set described in [29]. This data-set [29] currently is the largest code-first Web Services data-set in the literature. An important feature of this data-set is that the services came from real-life open source projects that actually implement Web Services or EJBs that were servified. The projects were gathered from different sources, namely Google Code, Exemplar [17] and Merobase[i].

We performed three separated experiments. The first one, which is discussed in Section 4.1, was designed to measure the effectiveness of our heuristics for automatically detecting potential anti-pattern causes in the Web Services source code. The second experiment, which is presented in Section 4.2, had two goals. The first goal was to determine how our tool performs in terms of anti-patterns avoidance when compared to a well-known code-first Web Service development tool, namely Java2-WSDL, which is part of Apache Axis2 [j]. The other goal was to determine whether the refactorings proposed by our tool benefit the quality of the generated WSDL documents. To do so we compared the WSDL documents generated with the original version of the Java projects against the WSDL documents generated with an improved version of the Java projects obtained by applying our suggested refactorings. The third experiment is presented in Section 4.3 and compares our approach against the OO metrics based approach to anti-pattern prevention in [29]. Finally, Section 4.4 discusses and summarizes the implications of the three separated experiments in order to provide a general view of the results.

4.1 Automatic detecting discoverability issues evaluation

Methodologically, each class that is mapped to a Web Service was manually analyzed looking for anti-pattern causes. Since different anti-patterns affect different portions 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 detector effectiveness, 568 parts were analyzed, but 1,563 parts were analyzed for detecting *Whatever types* anti-pattern causes. Table 4 shows which parts are analyzed by each detector and the quantity of them in the employed data-set. Notice that the analysis of the Inappropriate and lacking comment detector experiment consists of two separated validations: *Lacking comments* causes detection and *Inappropriate comments* causes detection. Also, the *Ambiguous names* causes detector experiment has been validated by means of validating the sub-cause detectors, namely *Ambiguous names in operations* and *Ambiguous names in parameters* . This is because the procedures for detecting potential issues in method names and parameter names are slightly different.

[i] Merobase: http://merobase.com
[j] Axis 2 Java: http://axis.apache.org/axis2/java/core/

Detector	Analyzed parts	Number of parts
Inappropriate or lacking comments	All public methods	568
Ambiguous names in operations	English named public methods	556
Ambiguous names in parameters	English named parameters of public methods	986
Low cohesive operations in the same port-type	All parameters and return values of public methods	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 4. Detectors: Analyzed parts per service class

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			
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	**Negative**	1,456	8	1,464	98.83%	0.55%	12.82%
	Positive	10	68	78			
Ambiguous names in operations	**Negative**	504	4	508	97.66%	0.79%	18.75%
	Positive	9	39	48			
Ambiguous names in parameters	**Negative**	952	4	956	99.49%	0.42%	3.33%
	Positive	1	29	30			
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			
				Average	**98.56%**	**0.29%**	**7.06%**

Table 5. Confusion matrices for the various detectors

The results from the manual analysis were compared with the results of executing the detection heuristics on the projects. After that, we constructed a confusion matrix for each detection heuristic. Table 5 presents the different confusion matrices as well as the average accuracies, false positive rates, and false negative rates of the detection heuristics. Notice that this average only considers the full-detectors and not theirs parts, e.g., the average considers the values for *Inappropriate or lacking comments* causes detector, but not the individual values of *Inappropriate comments* and *Lacking comments* detectors. In this table, the main anti-pattern detectors are marked with gray cells, while the sub-heuristics are in white cells.

To avoid possible bias, the manual analysis was carried out by two final-year System Engineering students, who had knowledge on Service Oriented Computing concepts, code-first Web Service development and WSDL anti-patterns. However, the students did not have knowledge on the internals of our plug-in nor its detection heuristics. Initially, the students were provided with the anti-pattern catalog presented in [34]. Then they were instructed to look for similar issues that the ones described by the anti-patterns presented in Table 4 in the classes of the different projects to be mapped to WSDL documents. For doing that, they had to identify each code element that might be affected by each anti-pattern and to decide whether an element was affected. We also instructed them to divide the *Inappropriate or lacking comments* anti-pattern into the sub anti-patterns *Inappropriate comments* and *Lacking comments* . In addition, they also had to separate the *Ambiguous names* anti-pattern into *Ambiguous names in operations* and *Ambiguous names in parameters* sub anti-pattern. Since the projects in the data-set provide Web Services for commonplace application domains, there was no training regarding the project domains. The idea behind this was to avoid introducing bias. Finally, the students compared their results and discussed each disagreement to provide a final result list. These discussions also were unsupervised to prevent external bias.

On average, considering only the experiments for the main five detectors (i.e., those marked with gray in the table), 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 7.06%, respectively. In all the cases, the accuracy was never lower than 96%. However, the false positives rates were high for *Inappropriate comments* , *Ambiguous names* ,*Ambiguous names in operations* and *Low cohesive operations in the same port-type* anti-patterns, which means that these heuristics are too sensitive to some problems inherent to real-life source code. Since the goal of the heuristics is to spot potential problems to manually refactor the source code, it is more important to have low false negative rates rather than low false positive rate. Below we provide a more in-depth analysis of these results.

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. Yet, the *Inappropriate comments* causes detection heuristic did not worked as accurately as the other heuristics. It had an accuracy of 93.99% and a high rate of false positives. However, a main issue is that developers tend to not provide comments for classes and interfaces that represent APIs [20].

In the case of the detector for *Ambiguous names* , the analysis shows that the effectiveness of this heuristic is heavily impacted by the detection of the *Ambiguous names in operations* heuristic. Although the *Ambiguous names in operations* has a high false positive rate (18.75%), the absolute number of false positives is relatively low (only nine occurrences). In addition, the overall accuracy of the *Ambiguous names in operations* cause detection heuristic was 97.66%. In contrast, the experiments

for *Ambiguous names in parameters* cause detection heuristic reported lower false positive and false negative rates, which resulted in a higher accuracy. Consequently, in general the *Ambiguous names* causes detection heuristic worked fairly well.

The detector for *Low cohesive operations in the same port-type* anti-pattern causes consists on determining if all the methods described in a class or interface are functionally related. We have found that the accuracy of this detector was higher than 96%. However, the false positives are as high as 24.13%, which means that a developer should decide whether the anti-pattern is effectively affecting the class, because this requires knowledge on the functionality of the service at hand. Regarding false negatives, the highest rate is 0.92%, which means that the detectors rarely missed detecting 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, detectable causes of too generic types/classes. According to our data-set, Java developers tend to avoid using Object in method signatures. Furthermore, they usually take advantage of generics when defining lists and maps, which reduce the amount of code because, otherwise, castings would be required. Additionally, all the common Java IDEs, such as Eclipse, NetBeans[k]and IntelliJ IDEA[l], enforce generics usage.

Finally, there were no cases of the possible root causes of *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 produce false positive results. Yet, the fact that there were no cases of this means that the developers of the analyzed projects in the data-set tended to use correctly the Java error handling mechanism. Therefore, if the WSDL documents generated from this data-set would be affected by *Undercover fault information within standard messages* anti-pattern, it means that the problem is caused by the generation tools and not by the developers.

4.2 WSDL document generation evaluation

This experiment was intended to provide empirical evidence that our WSDL generation tool generates fewer anti-patterns that the well-known Java2WSDL generation tool. In addition, the experiment was also intended to show that if a developer applies the suggestions provided by our tool, the generated WSDL documents will have better quality than the WSDL documents generated from the original code in terms of anti-patterns occurrences.

To do this, we executed the automatic detection heuristics using as input the classes in the data-set that defines a service API. Then, the students, not having knowledge on the internals of our heuristics, refactored these classes in order to remove the detected issues. Table 6 briefly describes the specific refactoring used for each possible detected issue. These refactorings are better described in Section 3. Notice that ideally the automatic detector should not detect any issue in the refactored classes. However, in 4.44% of the cases, the detector continued reporting *Inappropriate comments* due to the detector false positives. In addition, there were two classes in which the logic of the Java code was so complex and the code so cryptic that the students were unable to refactor them.

After applying the refactorings, we had two data-sets, namely the original and the improved one. For each data-set, we constructed two sets of WSDL documents: one using Axis Java2WSDL and another one using our WSDL document generation tool. As a result, we generated 4 sets of WSDL documents: original+Java2WSDL, original+our tool, improved+Java2WSDL, and improved+our tool. In

[k] NetBeans: `https://netbeans.org/`
[l] IntelliJ IDEA: `https://www.jetbrains.com/idea/`

Detector	Refactoring
Lacking comments	Add comments to the method.
Inappropriate comments	Rewrite the comments to make them more descriptive.
Ambiguous names (method and parameter names)	Replace ambiguous names with non-ambiguous ones.
Low cohesive operations in the same port-type	Remove uncohesive operations from the class or revise their names/documentation.
Whatever types	Replace the 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 be thrown by the method.

Table 6. Applied refactorings

order to assess the quality of each WSDL document set, we used the automatic Anti-pattern detection tool [32, 35], following the same methodology as [29].

Figure 4 presents the experimental results in two subfigures to compare the number of occurrences of a particular anti-pattern in each set of WSDL documents:

- Figure 4 (top) depicts the absolute frequency of anti-patterns occurrences in the 4 WSDL document sets.
- Figure 4 (bottom) depicts the relative frequency of anti-patterns occurrences in the 4 WSDL document sets.

Note that since the number of WSDL element tags potentially affected by an anti-pattern depends on the WSDL document set and the anti-pattern, Figure 4 (bottom) allows comparing them relatively.

The first anti-pattern in the figures is *Lacking comments* , which affects all the WSDL documents generated using Java2WSDL. This is because Java2WSDL uses as input the Java bytecode of service classes, which contains no comments, to generate the WSDL files. Our tool, in contrast, maps the Java source to WSDL documents, and so the operations affected by no comments are the ones that map back to uncommented methods. In the case of the original data-set, our tool found no comments for 62% of the methods. In the improved version, a small subset of these methods still had no comments, which correspond to the two unrefactored, cryptic classes.

Regarding the *Inappropriate comments* sub anti-pattern, the WSDL documents generated using Java2WSDL did not present this anti-pattern because there were no comments. In contrast, the WSDL documents generated with our tool were affected by this anti-pattern. In particular, the anti-pattern affected 22.6% of the operations with comments in the WSDL documents generated with our tool from the original data-set, and 0.5% of the commented operations of the WSDL document generated from the improved data-set, which means 43 and 2 instances of the anti-pattern, respectively.

When analyzing the *Ambiguous names* anti-pattern, it can be seen that in both cases, original and improved, our tool generated WSDL documents with fewer ambiguous names than Java2WSDL. The relative frequency of this anti-pattern is slightly higher in the WSDL document generated with the improved data-set than the WSDL documents generated based on the original services. The difference is 2.01% for Java2WSDL and 0.94% for our tool. Although this is undesirable, the absolute number of occurrences diminished when using the improved data-set. Actually, the WSDL documents generated

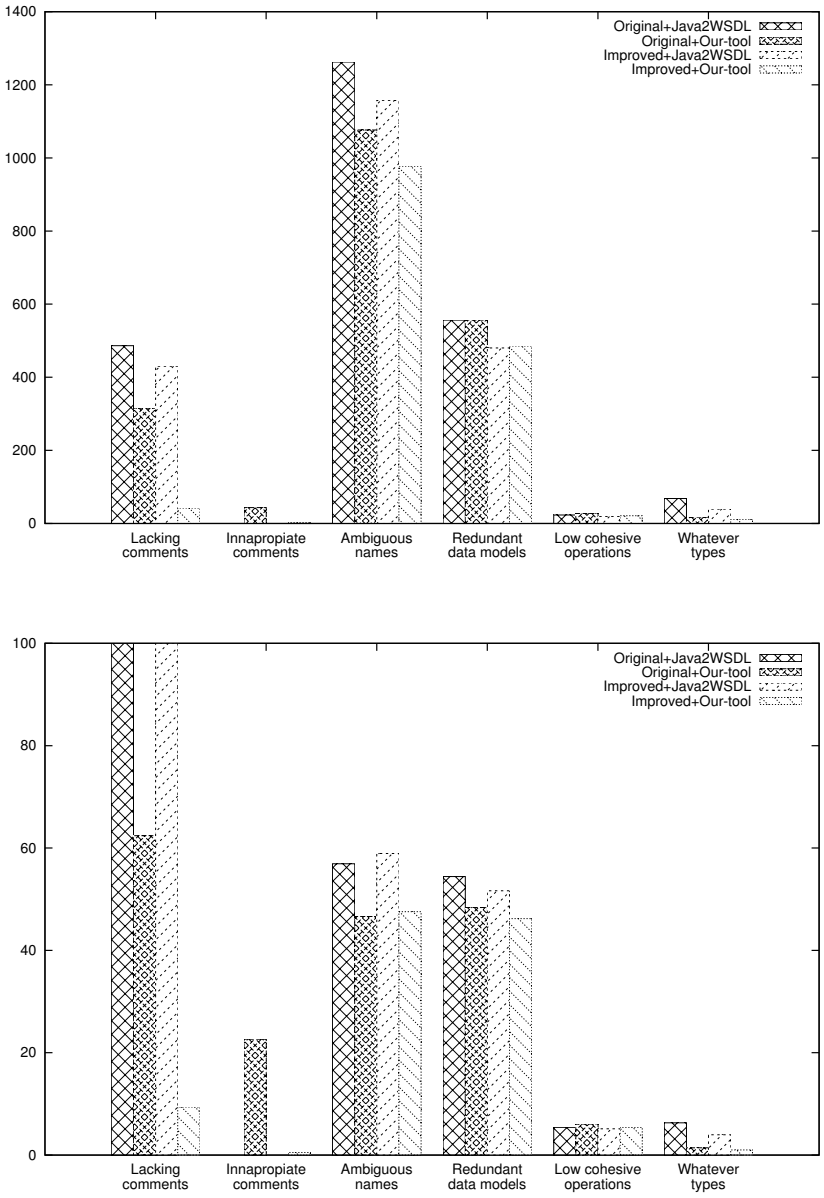

Fig. 3. Encountered anti-patterns in the generated WSDL documents: Absolute occurrences (top) and relative ocurrences (bottom)

using the improved data-set have 8.3% and 9.2%, for Java2WSDL and our tool respectively, fewer occurrences than when generating them using the original data-set. The slightly increase in the relative frequency partially stem from removing uncohesive methods from the classes. This refactoring removes names that are not ambiguous reducing the proportion of affected names.

Regarding the *Redundant data models* anti-pattern, in both cases, using the original data-set and the improved one, our tool reported slightly better results than Java2WSDL. In addition, the WSDL documents generated using the improved data-set defined less redundant data-types than the WSDL documents generated using the original data-set.

With regard to *Low cohesive operations in the same port-type* port-type, both tools reported similar results for each data-set. In general, the results are slightly better for the WSDL documents generated in the improved data-set. However, both tools performed the same because the generation process cannot avoid the low cohesive methods in the class that is being mapped.

The last anti-pattern considered in Figure 4 is the *Whatever types* anti-pattern. In this case, both tools generated better WSDL documents when using the improved data-set. In relative terms, Java2WSDL generated 6.3% of the types using whatever types, and our tool 1.4% when using the original data-set. These values were reduced to 3.9% and 0.99% when using the improved data-set. The higher value of Java2WSDL is explained by the fact that the tool also try to map objects provided by external libraries used by the services, and in doing so more occurrences of the anti-pattern arise. Our tool operates at the source code level, and as such only the occurrences present in service implementations are counted here. However, the reduction of the anti-pattern occurrences in both cases was 38% and 29% for Java2WSDL and our tool, respectively.

Finally, Figure 4 does not present any information related to the *Redundant port-types* anti-pattern and the *Undercover fault information within standard messages* anti-pattern, because there were no detected occurrences of them in the generated WSDL documents. In contrast, the *Enclosed data model* anti-pattern is not discussed since Java2WSDL does not allow separating the XSD from the WSDL document, while our tool does so, and then the comparison would be unfair.

The experiment in this subsection showed that out tool performed competitively when compared to Java2WSDL in regards to *Inappropriate or lacking comments* and *Ambiguous names* , which are the anti-patterns that most affect WSDL-based discovery [34]. In addition, our tool performed slightly better than Java2WSDL for the *Redundant data models* anti-pattern and the *Whatever types* anti-pattern, while our tool performed marginally worse for the Low cohesive operations within the same port-type anti-pattern. To sum up, our tool performed similarly to this well-known code-first generation tool for most of the anti-patterns, but notably better for the anti-patterns that affect Web Service discoverability the most.

4.3 Service code smells based vs OO metric based WSDL generation

This experiment aims at assessing the relative improvement of using our refactoring approach – together with our WSDL generation tool and Java2WSDL– versus the OO metric based approach in [29] and Java2WSDL. The methodology, which basically is the same as the one described in the previous section, consisted in refactoring the Web Service implementation data-set using both refactoring approaches, generating the WSDL documents using the generation tools –namely Java2WSDL and our tool– and then, counting anti-patterns occurrences using the anti-pattern detector described in [35]. Having the numbers of occurrences for each anti-pattern, we calculated the improvement for each anti-pattern as:

Approach	Tool used with the original service code	Tool used with the refactored service code	*Inappropriate or lacking comments*	*Ambiguous names*	*Low cohesive operations in the same port-type*	*Redundant data models*	*Whatever types*
OO metric based	Java2WSDL	Java2WSDL	0%	-8.49%	38.88%	-19.73%	100%
Code smells based	Our tool	Our tool	87.71%	9.01%	18.87%	8.06%	35.30%
	Java2WSDL	Java2WSDL	0%	8.16%	4.98%	10.09%	61.70%

Table 7. Relative improvements using different approaches and tools

$$\%improvement = \frac{OcOrig - OcRef}{OcOrig}$$

where, $OcRef$ are the number of occurrences for an anti-pattern in the refactored data-set, and $OcOrig$ are the number of occurrences for the same anti-pattern in the original (non-refactored) data-set. Moreover, for our approach, the *[Our tool, Our tool]*, *[Java2WSDL, Java2WSDL]* combinations were used to originate the *[Original, Refactored]* WSDL documents. Since we support two WSDL generation tools, the former combinations allowed us to better measure the impact of our refactorings compared to the refactorings proposed by the authors in [29].

Table 7 presents the results of this experiment. The Table does not show the *Undercover fault information within standard messages* anti-pattern because no occurrences were found in the data-sets. Besides, since the OO metric based approach focuses on studying service code refactorings rather than proposing a custom WSDL generation tool, we omit the anti-patterns that are caused at WSDL build time, i.e., *Enclosed data model* and *Redundant port-types* . Usually, the OO metric based approach is able to solve almost all occurrences of the *Enclosed data model* anti-pattern because the approach recommends removing all the complex data-types replacing them by primitive data-types. Regarding the code smell based approach, it is important to point out that our tool is able to create the schema in a separated file so the resulting WSDL documents would not be affected by this anti-pattern either. A similar situation occurs with the *Redundant port-types* anti-pattern, since our tool avoids this problem, whereas Java2WSDL does not.

With respect to the *Inappropriate or lacking comments* anti-pattern, [29] cannot improve services since its associated tool, Java2WSDL, does not include any documentation in the generated WSDL documents. Therefore all generated WSDL documents with this tool lack comments. Of course, the same applies to our approach and Java2WSDL. However, when using our WSDL generation tool, the amount of operations that have inappropriate comments or lack comments is reduced by 87.71%. This happens because our tool maps the method comments into operation comments. In consequence, if a method that performs an operation computation is properly commented, the operation would be properly commented.

First, the OO metric approach with Java2WSDL generates more ambiguous names than without refactoring the code. This undesirable effect is caused by a) the fact that this approach does not focus on text-related refactorings, and b) it encourages splitting a service with many operations into smaller services to increase their cohesion. Then, after WSDL generation, the original ambiguous names in services are further cloned. On the other hand, the code smells based approach using our tool reduces the *Ambiguous names* by 9.01% and by 8.16% when Java2WSDL is used instead of our tool. This is

important since as indicated in the previous subsection it is known that *Ambiguous names* is one of the anti-patterns that most affect WSDL legibility and discoverability [34].

Regarding the *Low cohesive operations in the same port-type* anti-pattern, the OO metric based approach had the best performance. However, this came at the cost of duplicating the number of WSDL documents generated by arbitrary dividing functionality. This leads to fine-grained services, resulting in the well-known Chatty Services problem[m] that has a negative effect on service consumer application performance. In addition, from a service consumer point of view, it is desirable for all the operations related to a coherent functionality to be in the same service, as this makes understanding and using such service easier. Coincidentally, this is a well-known good practice in general API design [5]. To sum up, dividing a complex service into two or more services reduces the number of uncohesive operations, but if this is done without care, it could lead to services that are inefficient and difficult to use.

When comparing *Low cohesive operations in the same port-type* anti-pattern impact using the code smells based approach, our tool presents a better performance than Java2WSDL. Since there are only two versions of the source code, e.g., the original and the refactored one, the number of uncohesive operations in each one should be the same independently of the used generation tool. The difference results then from the WSDL document anti-pattern detector [35] that uses the text in the operations to determine their domain. Since our tool generates WSDL documents with comments and better names, the results of the detector always benefits our tool. Taking this into account, the result that best assesses the impact of our approach is when our WSDL generation tool is used for both original and refactored services.

Using the OO metric approach negatively impacts on the number of *Redundant data models* anti-pattern occurrences. This happens because Java2WSDL generates types for wrapping the XSD primitive types, such as strings or integers, once for each type used. This results in several definitions of the same structure for strings, integers, booleans, etc. In contrast, when using both tools in conjunction with the refactored code via the anti-pattern causes based approach, the WSDL documents have less redundant data models. This is because the refactoring proposed by our approach is to eliminate the redundant data definitions rather than using primitive types. Although Java2WSDL could still create redundant wrapping for these data models when the same data-type is used more than once, the number of times that a particular data-type is used as input, output or fault message type is less. Furthermore, many data-types definitions are not used directly as messages, but indirectly to define other data-types.

The refactored code obtained with the OO metric approach does not have any whatever types because all the data-types were turn into primitive types, which is the suggested refactoring by that approach for dealing with redundant data-types. However, using a text string to convey complex information is a whatever type that the WSDL document anti-pattern detector cannot detect because this kind of analysis requires knowing the meaning of these strings, which clearly cannot be inferred from the WSDL document. When using the anti-pattern causes based improvements, the number of whatever data-types is reduced by 35.30% using our tool and 61.70% using Java2WSDL. The improvement with Java2WSDL is greater that the one obtained with our tool mainly because when using the original data-set Java2WSDL generated 47 data-types that use whatever types while our tool only generated 17.

[m] Chatty Services: http://www.ibm.com/developerworks/webservices/library/ws-antipatterns/

		Manually detected issues		Total	Accuracy	False negative	False positive
		Negative	Positive				
Automatically detected issues	Negative	4,127	12	4,139	98.95%	0.29%	6.23%
	Positive	38	572	610			
Total		4,165	584	4,749			

Table 8. General issues detection

All in all, taking into account the results presented in Sections 4.2 and 4.3, our approach together with our tool generate WSDL documents with less ambiguous names, and their operations always have comments and are likely to be well-commented, which is important for service discoverability [34]. Although the OO metric based approach generates more cohesive WSDL documents, it requires n-duplicating the number of WSDL documents. In open environments, this can dramatically increase the size of Web Service registries, forcing developers to invest even more effort into discovering required services. In addition, this approach tends to use strings for representing arbitrary data-types. This means that the whatever types are implemented using strings instead of generic objects. In the end, this practice makes Web Services difficult to be understood and discovered by third-parties. Finally, when using our approach and tool, the resulting WSDL documents do not have the *Enclosed data model* anti-pattern, since our tool supports exporting XSD data-type definitions into a separated file.

4.4 Discussion

According to the results presented in Section 4.1, i.e., the first set of experiments, the proposed heuristics can effectively detect potential issues. On average, the accuracy of the detectors was 98.56% when compared with the results of manually analyzing the Java projects. An important fact is that the false negative average rate was 0.29% because it is very likely that a developer using the tool misses an issue that was discarded by the heuristic. This is because we expect that developers would use our tool to narrow down which code parts they would inspect looking for potential issues. In contrast, the average rate of false positives was higher (7.06%). As a result of this, by relying on the suggestions of our tool a developer would be forced to analyze certain code sections that do not actually require to be refactored. However, intuitively, the time spent in doing this is negligible versus the time saved in manually spotting the actual code issues. This average was calculated as the arithmetic mean among the accuracies associated to the anti-pattern cause detectors, i.e., it excludes the sub anti-pattern causes detectors and does not consider the number of analyzed parts, which was depicted in Table 4.

Table 8 shows the performance of all the anti-pattern detectors in terms of the analyzed parts. This single confusion matrix was calculated as the sum of the anti-pattern cause detectors' confusion matrixes. This means that the total number of parts (4,749) is the total number of parts that were analyzed by considering all the detectors. By analyzing the results as a whole, it can be seen that only 12 parts were misclassified as not having issues. Mainly, if developers only analyze automatically detected issues, they would miss 2.05% of the issues. Furthermore, they would have to analyze 38 parts that have no issues, which means a 6.23% of the parts to analyze. A potential threat to validity of these results is that some issues might affect the accuracy of the other issues detection. For instance, the *Ambiguous names* anti-pattern causes potentially affects the *Low cohesive operations in the same port-type* and *Inappropriate comments* anti-patterns causes detection. This is because the latter detectors use parameter and operation names as part of the detection heuristic. However, the presented results show that, even when the proposed detection heuristics can be improved, the detected issues are a good starting

point for quickly identifying potential refactorings.

One of the main drawbacks of our approach to issue detection is that the heuristics for detecting *Inappropriate comments* and *Ambiguous names* in both methods and parameters are language dependent. In this work, the presented heuristics were designed for standard English. Therefore, it is impossible to assure that they would work as well in another language, or in other words extrapolate the results in Table 8. However, only three out of the seven detection heuristics heavily depend on the language. Besides, the language limitation is not exclusive to our heuristics, but it is a common problem in automatic code naming analysis [21]. This is because this kind of analysis relies on language corpora [26] and statistical properties [11]. Despite their limitations, the heuristics can detect issues that are well-known issues in API definitions [42, 20].

The experimental results presented in Section 4.2 confirm that removing the detected issues results in better quality WSDL documents, even when using Java2WSDL, the most used Java-based code-first tool to generate WSDL from service source code. Therefore, the experimental results point out that applying the refactorings is beneficial even though the WSDL document generation tool is not optimized to avoid anti-patterns. In most of the cases, the percentage of affected parts by certain anti-patterns was reduced independently of the generation tool. The only exception was the *Lacking comments* anti-pattern when using Java2WSDL. This is because Java2WSDL uses Java compiled code as input to generate WSDL documents. During the compilation process, the Java compiler ignores code comments, so the compiled code has no comments and hence Java2WSDL has no comments to place in the WSDL documents.

When compared against the OO metric approach to improve WSDL documents discoverability [29], our approach reduced the occurrences of all the anti-patterns. Despite improving discoverability, applying the OO metric approach [29] resulted in more occurrences of some anti-patterns. This is undesirable because having more anti-patterns makes it difficult for service consumer to invoke the services. The main advantage of our approach is that, compared to [29], it significantly reduces the occurrences of the *Inappropriate or lacking comments* anti-pattern. In this context, lacking and low quality comments are two well-known issues in APIs [42, 18]. Furthermore, using [29] increments the number of the *Ambiguous names* anti-pattern, while our approach reduces it.

In contrast, the OO metric approach [29] outperformed our approach in terms of the *Low cohesive operations in the same port-type* and the *Whatever types* anti-patterns. However, the refactorings that are proposed [29] have two main downsides. For reducing *Low cohesive operations in the same port-type* , the OO metric approach indicates that large classes have to be split, but it does not specify how. As a result, a large Web Service can end in the worst case as several one-operation Web Services. Moreover, although each resulting Web Service would be cohesive by definition, having several one-operation Web Services might not be the best way to organize them as a whole [31]. In this context, placing related operations in different Web Services makes it difficult for service consumers to identify operations of the same domain they might need. Regarding the *Whatever types* anti-pattern, the OO metric approach proposes to replace generic types, e.g., Object, with primitive types and strings. However, this conceals the issue from the detector because the string does not carry a piece of information that is supposed to be represented as a string. As a result, whoever decides to use the service needs to know the string format for successfully invoking the service.

Briefly, the proposed heuristics are capable of detecting different issues in Web Service Java source code that leads to anti-patterns in the generated WSDL documents. Additionally, the experimental results show that correcting these issues has a positive effect in the WSDL document quality by reducing

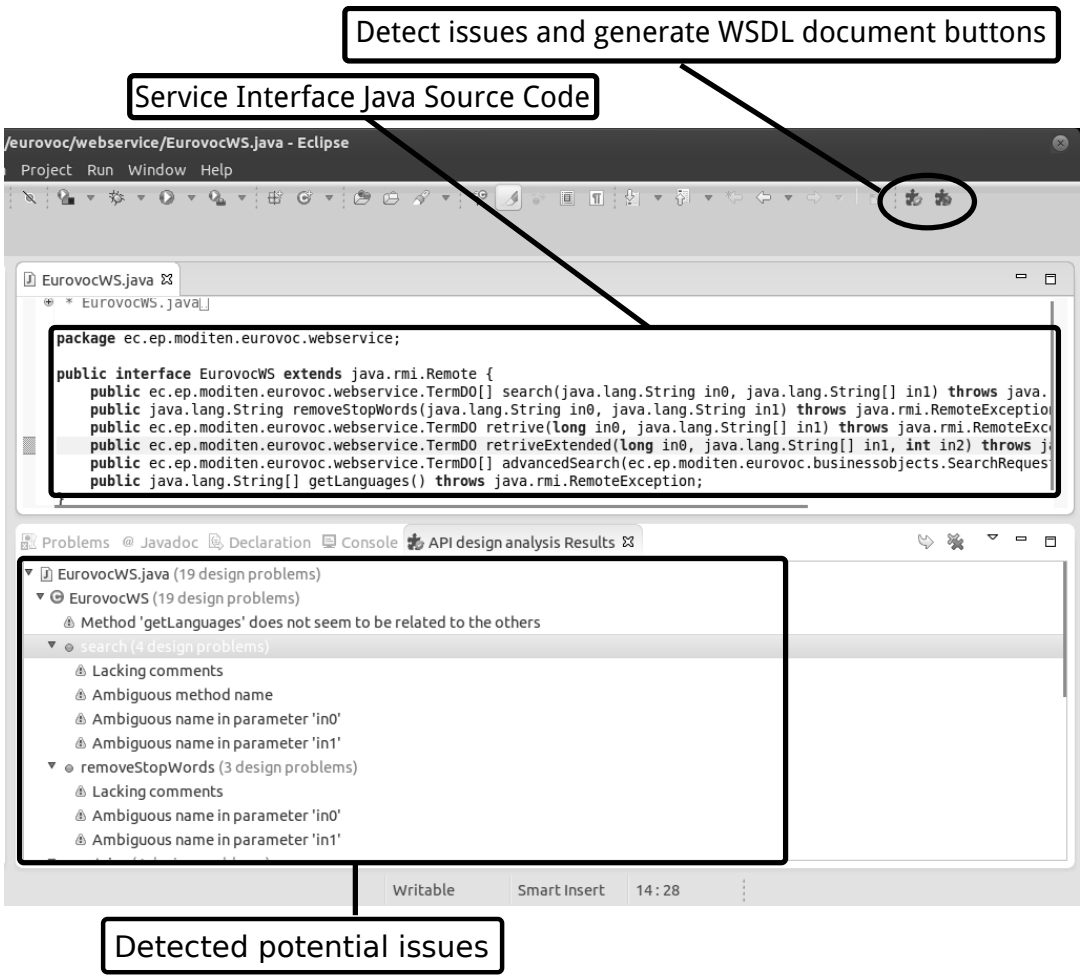

Fig. 4. Eclipse plug-in

the number of anti-pattern occurrences. This is in accordance with previous research [15, 3, 34, 33] showing that removing anti-patterns makes Web Services easier to be discovered and understood by service consumers. Even when service consumers might use WSDL documents to automatically generate –i.e., without manual WSDL inspection– stubs for invoking services, WSDL documents quality is important. Basically, the generated stubs have a direct relation with the WSDL documents [19]. Although further research is needed, API design problems can be transferred from WSDL documents to the automatically generated stubs. For instance, the stub generator would have only the WSDL document as source for method names and parameter types.

5 The Eclipse plug-in

Our detectors have been implemented in the form of a plug-in for Eclipse, which is available upon request. As it is shown in Figure 4, the plug-in has two buttons: one performs the anti-patterns cause detection and the other one launches the wizard for generating WSDL documents. The analysis is

performed over the current class in the main developer view, which is assumed to define a Web Service API. In the lower part of the screen, the plug-in reports the issues detected in the class source code. In addition, a tooltip that describes the issue and a suitable refactoring appears when the developer moves the mouse over the issue.

The tool also includes a WSDL generation wizard comprising two consecutive screens. 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 he/she wants to use and where the WSDL document file will be generated. Currently, the plug-in supports two tools: Axis' Java2WSDL and our tool (see Section 3.2), and we are working on incorporating EasyWSDL.[n] This screen also gives the developer the option of generating the WSDL document or going on to the second screen to configure the XSD file generation 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.

6 Conclusions

This work proposed an approach and tool to assist code-first Web Service developers to improve the quality of WSDL-described Web Services. Basically, our proposal is intended to detect problems in Web Services implementation code that existing WSDL generation tools might map to WSDL anti-patterns in the generated WSDL documents. Knowing the potential issues, developers can assess the WSDL document quality and apply suitable refactorings as needed, and thus generating better quality WSDL documents for his/her code-first Web Services. Our proposal also avoids anti-patterns that manifest upon WSDL generation and not only Web Service code programming.

Our work enforces practices that have been proved to be very important for a SOC system success [34]. In addition, it is known that if these practices are not taken into account during the first stages of service development, reengineering a system afterwards might be extremely costly [33]. Since it seems that SOC developers often disregard these practices [34, 29, 33], a tool for assisting them is necessary for improving a SOC system chance of success.

Our experiments show that the proposed detection heuristics performed fairly well for real-life open source Java Web Services. In all the cases, the false negative rate was less than 1%. Furthermore, the false negative rate was 0% for 2 out of 5 anti-pattern detectors. In contrast, some heuristics presented a high false positive rate. The worst case was the *Inappropriate comments* cause detector that reported a false positive rate of 24.13%. However, classifying operations by *Inappropriate comments* anti-pattern occurrences results in skewed classes. In our data-set, only 44 out of 233 were classified as *Inappropriate comments* anti-pattern cause, i.e., 18% of the comments had low quality. This means that even with a false positive rate of 24.13%, the developer would have to analyze 58 comments instead of 233. Moreover, since this detector had a low false negative rate, which was 0% in our experiments, it is unlikely that the developers would miss any occurrence of *Inappropriate comments* anti-pattern cause. In absolute values, our heuristics never returned more than 20 false positives in data-sets of hundreds or thousands of elements, thus effectively narrowing the amount of elements that developers must manually analyze.

Secondly, we assessed the impact of removing the detected anti-pattern causes from Java code on the generated WSDL documents. To provide a baseline, we generated the WSDL documents

[n] EasyWSDL: `http://easywsdl.ow2.org/`

for the original Java codes using the well-known tool Java2WSDL. Furthermore, we also generated the WSDL documents for the original Java codes using our tool. Then, we generated the WSDL documents for the improved version of the codes using both tools. This resulted in four WSDL document sets that were analyzed using the WSDL document anti-pattern detector [35]. According to the results, our tool outperformed Java2WSDL in four out of the five anti-patterns analyzed, i.e., the anti-patters whose cause is detected by our heuristics, and both tools performed fairly similar in the case of the *Low cohesive operations in the same port-type* anti-pattern. These results are independent of whether the Java source code was refactored. Finally, the improved Java code produced WSDL documents with fewer anti-pattern occurrences when using our generation tool. However, refactoring the Java code did not always resulted in fewer anti-patterns. For instance, since Java2WSDL ignores comments in the source code, using it always results in *Inappropriate or lacking comments* anti-pattern occurrences independently of the comment quality and quantity in the Java source code.

Finally, we compared our approach against the OO metrics based approach presented in [29]. The results indicated that our approach reduced the number of occurrences of the five anti-patterns it detects. In contrast, the OO metrics based approach obtained mixed results. Notably, the OO metrics based approach outperformed our approach for the *Low cohesive operations in the same port-type* and *Whatever types* anti-patterns. However, this is not necessarily a good result. In the case of the *Low cohesive operations in the same port-type* anti-pattern, the reduction is given because the OO metrics based approach tends to generate one-operation Web Services. A one-operation Web Service is never affected by the *Low cohesive operations in the same port-type* anti-pattern, but highly related operations are separated into different Web Services, which is not recommendable, since this practice fragments service functionality. For the *Whatever types* anti-pattern, the OO metric based approach recommends to change all data-types for Strings or primitive-types. Therefore, the *Whatever types* anti-pattern occurrences are masked by the use of arbitrary Strings that must be parsed in an ad-hoc manner.

7 Future work

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* cause's detector. For example, we are planning to enhance it using concept disambiguation. To do this, we are considering tools such as WikipediaMiner[o], a tool to assess how terms and concepts are connected to each other based on content published in Wikipedia [27]. In addition, we are extending the text-related detectors for supporting other languages apart from English. We will also extend our WSDL document generation tool to support other programming languages, such as C#. Another pending analysis is whether an incremental approach would improve the detection and refactoring process. For instance, starting by detecting and refactoring *Ambiguous names* anti-pattern causes first might lead to better detection of *Low cohesive operations in the same port-type* and *Inappropriate comments* anti-pattern causes. Finally, we will explore and eventually develop heuristics for *automatically* refactoring Web Services source code to diminish the number of anti-patterns generated by the WSDL generation tools.

Another research opportunity is deeply analyzing how discoverable the WSDL documents generated following our tool guidelines are and comparing them against WSDL document following the other guidelines based on the original anti-pattern definition [29]. In this context, discoverability can be measured by means of classical Information Retrieval metrics (e.g., precision and recall) and Web

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

Service search engines were original and refactored WSDL documents are published. The goal of this comparison is to assess which is the benefit of attacking the anti-pattern issue directly as we do, instead of indirectly as [29], in terms of these specific metrics. This could eventually lead to a hybrid approach to code-first service refactoring for WSDL anti-pattern prevention, i.e., combine the best of both approaches.

We are also 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 only understood as finding a Web Service, but finding who provides a particular Web Service. One example of this context are Home Services [40], 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[p]. Another similar usage context is when several Web Services are composed to provide more complex functionality. Since Web Services Quality of Service (QoS) might vary, it is important to replace Web Services with low QoS for Web Services with high QoS [41, 22, 40] to ensure a good global QoS.

Finally, in this paper, we have dealt with "syntactic" Web Services, as opposed to semantic Web Services, in which service interfaces are enriched with concepts extracted from shared ontologies. In this line, we aim at adapting the ideas presented in this paper in respect to WSDL specification, particularly the concept of WSDL anti-pattern, to semantic Web Services. The cornerstone of this line of research is the automatic approach to annotate Web Services with semantic data described in [9].

Acknowledgements

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

References

1. E. E. Allen and R. Cartwright. Safe instantiation in generic java. *Science of Computer Programming*, 59(1-2):26 – 37, 2006. Special Issue on Principles and Practices of Programming in Java (PPPJ 2004).
2. V. Basili, L. Briand, et al. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751 –761, oct 1996.
3. J. Beaton, S. Y. Jeong, et al. Usability challenges for enterprise service-oriented architecture APIs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 193–196. Sept. 2008.
4. M. B. Blake and M. F. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12(5):62–69, 2008.
5. J. Bloch. How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pp. 506–507. ACM, New York, NY, USA, 2006.
6. H. S. Chae, Y. R. Kwon, et al. A cohesion measure for object-oriented classes. *Software: Practice and Experience*, 30(12):1405–1431, 2000.
7. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
8. M. Crasso, J. M. Rodriguez, et al. Revising WSDL Documents: Why and how. *IEEE Internet Computing*, 14:48–56, October-September 2010.
9. M. Crasso, A. Zunino, et al. Combining document classification and ontology alignment for semantically enriching Web Services. *New Generation Computing*, 28:371–403, 2010.

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

10. —. A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management*, 22:103–134, 2011.
11. C. D. M. Dan Klein. Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, pp. 423–430. 2003.
12. F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.
13. T. Erl. *SOA Principles of Service Design*. Prentice Hall, 2007.
14. A. Erradi and P. Maheshwari. Enhancing web services performance using adaptive quality of service management. In B. Benatallah, F. Casati, et al., eds., *Web Information Systems Engineering (WISE 2007)*, vol. 4831 of *Lecture Notes in Computer Science*, pp. 349–360. Springer Berlin Heidelberg, 2007.
15. J. Fan and S. Kambhampati. A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32, 2005.
16. J. M. García, D. Ruiz, et al. Improving semantic Web Services discovery using SPARQL-based repository filtering. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17(0):12 – 24, 2012.
17. M. Grechanik, C. Fu, et al. A search engine for finding highly relevant applications. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10), Cape Town, South Africa*, pp. 475–484. ACM Press, New York, NY, USA, 2010.
18. T. Grill, O. Polacek, et al. Methods towards api usability: A structural analysis of usability problem categories. In M. Winckler, P. Forbrig, et al., eds., *Human-Centered Software Engineering*, vol. 7623 of *Lecture Notes in Computer Science*, pp. 164–180. Springer Berlin Heidelberg, 2012.
19. S. Hallé, G. Hughes, et al. Generating interface grammars from WSDL for automated verification of web services. In L. Baresi, C.-H. Chi, et al., eds., *Service-Oriented Computing*, vol. 5900 of *Lecture Notes in Computer Science*, pp. 516–530. Springer Berlin Heidelberg, 2009.
20. M. Henning. API design matters. *Communications of the ACM*, 52(5):46–56, May 2009.
21. E. Høst and B. Østvold. Debugging method names. In S. Drossopoulou, ed., *ECOOP 2009 - Object-Oriented Programming*, vol. 5653 of *Lecture Notes in Computer Science*, pp. 294–317. Springer Berlin Heidelberg, 2009.
22. W. Jiang, T. Wu, et al. Qos-aware automatic service composition: A graph view. *Journal of Computer Science and Technology*, 26(5):837–853, 2011.
23. N. Khamis, R. Witte, et al. Automatic quality assessment of source code comments: The javadocminer. In C. Hopfe, Y. Rezgui, et al., eds., *Natural Language Processing and Information Systems*, vol. 6177 of *Lecture Notes in Computer Science*, pp. 68–79. Springer Berlin Heidelberg, 2010.
24. D. Lawrie, H. Feild, et al. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.
25. S.-Y. Lin, C.-H. Lai, et al. A trustworthy qos-based collaborative filtering approach for web service discovery. *Journal of Systems and Software*, 93:217–228, 2014.
26. G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, Nov. 1995.
27. D. Milne and I. H. Witten. An open-source toolkit for mining wikipedia. *Artificial Intelligence*, 194:222 – 239, 2013.
28. T. Narock, V. Yoon, et al. A Provenance-Based Approach to Semantic Web Service Description and Discovery. *Decision Support Systems*, 2014. In press.
29. J. L. Ordiales Coscia, C. Mateos, et al. Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools. *International Journal of Web and Grid Services*, 9(2):107–126, 2013.
30. M. Papazoglou, P. Traverso, et al. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.
31. G. M. Rama and A. Kak. Some structural measures of api usability. *Software: Practice and Experience*, p. In press, 2013.
32. J. Rodriguez, M. Crasso, et al. Automatically detecting opportunities for Web Service Descriptions improvement. In W. Cellary and E. Estevez, eds., *Software Services for e-World*, vol. 341 of *IFIP Advances in Information and Communication Technology*, pp. 139–150. Springer Boston, 2010.
33. —. Bottom-up and top-down cobol system migration to web services: An experience report. *IEEE Internet Computing*, 17(2):44–51, 2013.

34. J. M. Rodriguez, M. Crasso, et al. Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001 – 1021, 2010.
35. —. An approach for web service discoverability anti-patterns detection. *Journal of Web Engineering*, 12(1-2):131–158, 2013.
36. M. Sharifi, S. B. Ramezani, et al. Predictive self-healing of web services using health score. *Journal of Web Engineering*, 11(1):79–92, Mar. 2012.
37. D. Steidl, B. Hummel, et al. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 83–92. May 2013.
38. T. Suzumura, T. Takase, et al. Optimizing web services performance by differential deserialization. In *2005 IEEE International Conference on Web Services*, pp. 185–192. july 2005.
39. T. Tenny. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, Sep 1988.
40. M. Wei, J. Xu, et al. Ontology-based home service model. *Computer Science and Information Systems*, 9(2):813 – 838, 2012.
41. M.-W. Zhang, B. Zhang, et al. Web service composition based on qos rules. *Journal of Computer Science and Technology*, 25(6):1143–1156, 2010.
42. M. Zibran, F. Eishita, et al. Useful, but usable? factors affecting the usability of apis. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pp. 151–155. Oct 2011.
43. G. Zou, Y. Gan, et al. Dynamic composition of web services using efficient planners in large-scale service repository. *Knowledge-Based Systems*, 62:98–112, 2014.