SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 9, no. 1, pp. 5-23 (2010)

An analysis of frequent ways of making undiscoverable Web Service descriptions

Juan Manuel Rodriguez[1] Marco Crasso[1] Alejandro Zunino[1] Marcelo Campo[1]

[1] ISISTAN Research Institute.
Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN)
Campus Universitario, Tandil (B7001BBO)
Buenos Aires, Argentina
e-mail: [jmrodri|mcrasso|azunino|mcampo]@exa.unicen.edu.ar
No. tel. 54-2293-440363, ext. 35

Abstract

The ever increasing number of publicly available Web Services makes standard-compliant service registries one of the essential tools to service-oriented application developers. Previous works have shown that the descriptiveness of published service descriptions is important from the point of view of the algorithms that support service discovery using this kind of registries as well as human developers, who have the final word on which discovered service is more appropriate. This paper presents a catalog of frequent bad practices in the creation of Web Service descriptions that attempt against their chances of being discovered, along with novel practical solutions to them. Additionally, the paper presents empirical evaluations that corroborated the benefits of the proposed solutions. These anti-patterns will help service publishers avoid common discoverability problems and improve existing service descriptions.

Keywords: Web Services; Web Service discoverability anti-patterns.

1 Introduction

The software industry is shifting from developing specific functionality from scratch to discovering, and combining functionalities offered by third-parties. Service-oriented computing (SOC) is a new paradigm for building software systems, in which developers look for software pieces, called *services*, within specialized registries to form their applications [Erickson and Siau, 2008]. Providers can use registries to advertise their services, while consumers can use registries to discover services that match their needs. The SOC paradigm promotes standard methods for remotely consuming services regardless of the technology. Accordingly, SOC enables succeeding in software development in heterogeneous distributed environments [Erickson and Siau, 2008] and, at the same time, promises reduction of coupling between components, agility to respond to changes in requirements, transparent distributed computing, and lower ongoing investments [McConnell, 2006].

The SOC paradigm is mostly implemented by using Web Services [Crasso et al., 2010]. A Web Service is a software system that can be discovered, and invoked using standard Web protocols. The growth of the Internet has popularized large repositories of Web Services [Bichler and Lin, 2006]. Unfortunately, finding proper services is very challenging, mainly because, unlike traditional software libraries, Web Service repositories rely on little meta-data for supporting service discovery [Sabou and Pan, 2007] and the ever increasing number of published Web Services [Garofalakis et al., 2006]. In the end, these challenges hinder the adoption of the SOC paradigm [Hummel et al., 2008].

A starting point to understand the challenges associated with current discovery methods is Universal Description, Discovery and Integration (UDDI)[1], a specification of standardized formats for programmatic business and service discovery, so that search engines could be built on top of it. With UDDI, publishing services consists in supplying a registry with the information associated with providers and technical descriptions of the functionality of their services in Web Service Description Language (WSDL)[2]. WSDL is an XML-based language for describing a service intended functionality by using an interface with methods and arguments, in object-oriented terminology, and documentation as textual comments. On the other hand, discoverers may look for third-party services by performing keyword-based searches.

Mostly because of the inconsistency of keywords in interfaces of publicly available services and queries [Paolucci and Sycara, 2003; Blake and Nowlan, 2008], efficiently finding proper services through implementations of UDDI is rather difficult [Garofalakis et al., 2006]. Nowadays, three main directions have been proposed to cope with this problem. As service descriptions usually reside in Web servers, one direction proposes to exploit the capabilities of Web search engines (e.g., Google) to crawl and index Web servers content [Song et al., 2007; Al-Masri and Mahmoud, 2008]. Although this approach is transparent to publishers, several studies have empirically shown that the precision of Web search engines when looking for known services does not significantly improve, even when proper comments had been introduced in the indexed WSDL documents [Song et al., 2007]. Accordingly, Web Service discoverers experience the same problems as ordinary users of Web search engines [Nakamura et al., 2007].

Another direction proposes to adapt Information Retrieval (IR) techniques, such as word sense disambiguation, stop-words removal and stemming, for extracting relevant words conveyed within WSDL documents, including their natural language comments, and, in turn, syntactically matching them against bags of words representing queries [Blake and Nowlan, 2008; Dong et al., 2004; Stroulia and Wang, 2005; Crasso et al., 2008b; Crasso et al., 2008a]. The main disadvantage of this direction is its inability to deal with meaningless service descriptions, i.e., lacking the information necessary for humans to understand the offered functionality, and poorly descriptive queries, e.g., when publishers use “arg0”, “foo”, for naming parameters and operations.

The two directions described previously, aim at exploiting service descriptions as they are, i.e., being Web Service standards compliant approaches to service discovery. Instead, the Semantic Web effort proposes to enhance service descriptions by annotating service descriptions with unambiguous concept definitions from shared ontologies [McIlraith et al., 2001; Sabou and Pan, 2007]. By assuming that all the aspects surrounding services are precisely described, it is expected that finding them will be simplified. However, semantic Web Services have not yet been adopted by the industry, since the effort needed to build such a semantic infrastructure is huge [McCool, 2006]. Obviously, syntactic approaches cannot replace the need for semantic machine-interpretable descriptions in the context of automatic applications, which discover services without human intervention [Paolucci and Sycara, 2003; Blake and Nowlan, 2008]. However, several studies [Dong et al., 2004; Crasso et al., 2008b; Crasso et al., 2008a] have empirically shown that syntactic approaches can effectively ease human discoverers’ tasks in practice without neither requiring all the specifications of full semantic techniques nor suffering from their known problems, namely the lack of standard ontologies and the high complexity involved in building them [McCool, 2006]. In the rest of this paper we will focus on approaches where service publishers and discoverers are human beings, instead of computer programs as most semantic approaches do.

Unfortunately, despite the fact that the rigorous experimental evaluations of IR-based approaches to service discovery have shown promising results [Dong et al., 2004; Crasso et al., 2008b; Crasso et al., 2008a], these results

1 UDDI http://uddi.xml.org/
2 WSDL http://www.w3.org/TR/wsdl

cannot be generalized, and they may vary with different data-sets. In fact, as the underpinnings of syntactic approaches to service discovery rely on the descriptiveness of service specifications, WSDL documents without proper comments or explanatory keywords may deteriorate discovery effectiveness. Furthermore, a poorly written WSDL document, besides having few chances of being properly retrieved by a registry, hinders human discoverers' ability to understand and select the service afterward. Understandability is crucial for a proper service discovery because if the human discoverer does not understand what a service does, they would not select the service. In other words, a registry returning incomprehensible services is equivalent to returning an empty list of services.

In spite of the intuitive implications of the use of non descriptive WSDL documents against syntactic approaches, to our knowledge, there is a lack of studies that identify, measure, and provide solutions to this problem. On the other hand, there is ongoing research on measuring the cost and benefits of bad and good programming guidelines, styles and API design practices [deSouza et al., 2004; Henning, 2009]. However, software practitioners lack empirical evidence showing whether these detected frequent practices occur in Web Services development or not, and whether these practices affect the discoverability of services.

To help publishers in creating services that can be easily discovered through syntactical and standard-compliant approaches, we have studied common practices found in a corpus of real world Web Service descriptions written in WSDL version 1.1 that, paradoxically, may attempt against service discoverability. This paper describes how to solve these frequent problems in a novel catalog of Web Service discoverability anti-patterns, which subsumes previous work on improving WSDL documents [Fan and Kambhampati, 2005; Pasley, 2006; Blake and Nowlan, 2008; Beaton et al., 2008], points out even more bad practices and proposes a reproducible solution to each identified bad practice.

To corroborate the utility of this catalog, we have compared the retrieval effectiveness of a syntactic discovery system using the original corpus of WSDL documents versus using the one that resulted after manually correcting found anti-patterns. Experimental results empirically show that the employed syntactic registry has a better performance discovering corrected WSDL documents. Additionally, we collected the opinions from 26 software engineering students and practitioners about service descriptions intelligibility. The collected opinions suggest that corrected WSDL documents may increase service chances of being understood and, in turn, outsourced. To sum up, the main contributions of this paper are:

1. a catalog of discoverability anti-patterns found in Web Service descriptions, which allows publishers to create more discoverable service descriptions or improve existing ones, and

2. experiments showing that employing this catalog to remove anti-patterns from WSDL documents is beneficial to connecting publishers and discoverers.

The rest of this paper is organized as follows. The next section presents details about WSDL documents, syntactic approaches for discovering services, and discusses related works. Then, Section 3 presents WSDL anti-patterns. Later, in Section 4 and Section 5 we survey the presence of anti-patterns in real Web Services and present a case study of how to apply the anti-pattern remedies in a real-life WSDL document, respectively. Following, Section 6 shows not only the implications of solving the identified anti-patterns when using a syntactic registry, but also how they hinder human discovers ability to understand services described by WSDL documents. Lastly, Section 7 concludes the paper and highlights future research directions.

2 Background and Related Work

Discoverers use functional descriptions in WSDL to match third-party services against their needs. WSDL is a language that allows developers to describe service functionality as a set of *port-types*, which arrange different *operations*, whose invocation is based on *message* exchange. Messages can either transport XML data between consumers and providers of services, and vice-versa, or represent exceptions (or faults in WSDL terminology). Optionally, each part of a WSDL file may contain comments in natural language.

Figure 1 shows the InfoSet of WSDL version 1.1. It is worth noting that WSDL documents have another element, called *bindings*. This, besides specifying technological aspects, such as transport protocol and network address (a.k.a. end-points), allows discoverers to know how to invoke selected services. For the sake of clarity, in Figure 1 and the rest of this section we focus upon explaining the part of a WSDL document that reveals the service interface that is offered to the outer world, and omit details of the *bindings*.

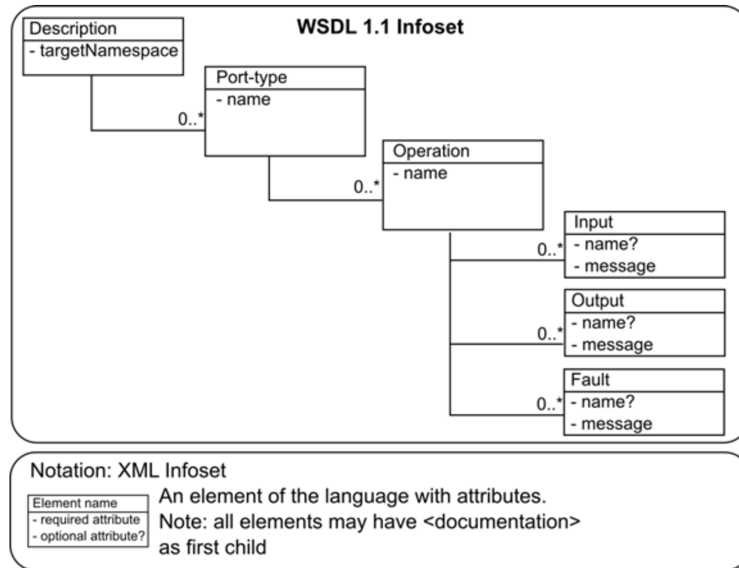

Figure 1: WSDL 1.1 InfoSet.

Messages comprise parts that transport structured data. Exchanged information is formed according to specific data-type definitions using the XML Schema Definition (XSD)[3], which is a language to express the structure of XML content. Figure 2 depicts a concrete WSDL document that contains the code needed for representing a complex data-type, called CountryCodes. The XSD definitions might be put into a separate file, and imported from one or more WSDL documents afterward.

In conclusion, WSDL is a language that allows developers to describe the offered interface of a Web Service using XML. Syntactic service registries commonly preprocess WSDL documents for extracting bags of words. During a discoverer's request, syntactic registries match the words extracted from each published service against the words conveyed within the query. Afterward, a rank of services is built according to the similarity among the words that were extracted from published services and the query. Finally, the user who performs the discovery must analyze the retrieved rank.

Then, in addition to the well-known advantages of documenting software and using explanatory names [Pendharkar and Rodger, 2002], the retrieval effectiveness of syntactic registries heavily depends on how effective for discovery the WSDL documents and queries are [Dong et al., 2004; Stroulia and Wang, 2005; Crasso et al., 2008b; Crasso et al., 2008a]. Therefore, the quality of WSDL descriptions is crucial for syntactical approaches to Web Service discovery, and in turn, for providers to get their services selected by human discoverers.

3 XSD http://www.w3.org/XML/Schema

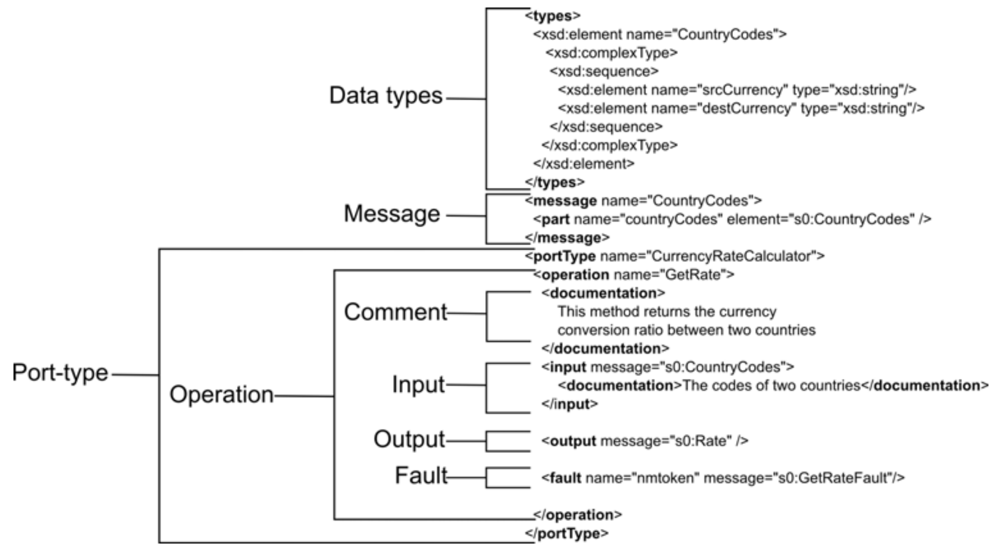

Figure 2: Example of Web Service description using WSDL.

Several efforts address the problem associated with the quality of WSDL documents from the perspective of discovery through syntactic service registries. Regarding documentation present in WSDL documents, our findings are supported by the study made by Fan and Kambhampati [Fan and Kambhampati, 2005]. Their study shows that in the 80% of 640 real-life WSDL documents, the average documentation length for operations is 10 words or less. Furthermore, from this 80%, half of them have no documentation at all.

In [Pasley, 2006] the author explains the impact of using XSD wild-cards on the maintainability and discoverability of Web Services. A wild-card is a special XSD constructor, which allows developers to leave undefined one or more parts of an XML structure. If a service message is related to a wild-card, then it will be able to transport either built-in types, such as xsd:string or xsd:long, or user-defined ones, e.g., “PayOrder” or “DatabaseRecord”. Clearly, this kind of definition does not allow discoverers to infer exactly what the input/response of a service is like. Although using wild-cards creates vague interface contracts, they are often used to minimize the effort involved in modifying a service when it evolves, while assuring that consumers bound to old versions of the service will be able to invoke correctly and process the operations defined in its new version [Pasley, 2006].

In [Beaton et al., 2008] the authors present many difficulties that six students of a SOC course encountered while developing a large Web Services-based application. Some of the identified difficulties are related to discovering third-party descriptions. Specifically, the authors show that unclear “control parameters” within data structures and long identifier names makes Web Services harder to be discovered [Beaton et al., 2008]. A message associated with a control parameter, besides carrying data objects, includes miscellaneous objects, such as the description of an error that occurs during the invocation of the operation. The authors refer to control parameter as a parameter value that may influence the execution flow of the service client.

Finally, in [Blake and Nowlan, 2008] the authors detect naming tendencies in WSDL documents and empirically show that the performance of a syntactic registry for discovering relevant services can be improved by enhancing its underlying matching approach for dealing with the observed tendencies. Broadly, the authors showed that developers use common phrases within parameter part names, abbreviations and names shorter than three characters, which were ineffective for matching part names.

In this paper we present a novel catalog of nine bad practices that frequently occur in a corpus of WSDL documents and may negatively impact on the precision of syntactic registries. This catalog not only subsumes the problems related to inappropriate or lacking comments [Fan and Kambhampati, 2005], XSD wild-cards [Pasley, 2006], control parameters [Beaton et al., 2008] and naming tendencies [Blake and Nowlan, 2008], but also supplies each problem with a practical solution.

3 A catalog of Discoverability Anti-Patterns

This paper explicitly addresses the quality of WSDL documents from the perspective of discovery, pursuing recurring problems that attempt against the understandability and discoverability of a service. It catalogs nine common bad practices that frequently occur in a corpus of WSDL documents that were gathered from Internet repositories by Hess et al. [Heß et al., 2004]. Each bad practice has been studied to provide a sound, and practical solution. In software engineering, a pattern is a general reusable solution to a commonly occurring problem. On the other hand, anti-patterns express obvious, and wrong solutions to recurring problems that have been supplied with refactored solutions that are clearly documented, proved and repeatable. According to the definition of an anti-pattern, Table 1 shows a comprehensive list of WSDL discoverability anti-patterns. Table 1 has six columns to present the next information:

- **Anti-pattern**: a descriptive name for the anti-pattern.
- **Concerns**: a classification of the anti-pattern based on how it affects a WSDL document. There are three types of concerns, "Documentation", "Design" and "Representation". The first type of anti-patterns is related to problems in the natural language description of the service. If the concern of the anti-pattern is "Design", it means that it is related to how the service interface is presented. Finally, anti-patterns classified as "Representation" refer to problems on how the real objects of the services are modeled with the data structures defined by the WSDL document.
- **Symptoms**: a brief description of when the anti-pattern occurs and what causes it.
- **Manifests**: a classification of the anti-pattern based on how it manifests itself. We refer to "Evident" anti-patterns as those that are present in the structure, or syntax, of a WSDL document. We refer to "Not immediately apparent" anti-patterns as those that require not only to analyze the syntax of a document, but also its semantics, to detect them. We refer to "Present in the implementation" anti-patterns as those that require invoking the service to detect them. This is further explained in Section 6.
- **Problems**: an enumeration of the discoverability problems the anti-pattern can cause.
- **Remedy**: a list of steps to follow in order to avoid the problems that may be caused by the anti-pattern.

As described in Section 2, syntactic registries require words from the WSDL document describing a service to index it. This kind of registries operates by extracting words, which represent the offered functionality of a service, from the corresponding WSDL document and creating a bag of words. However, some frequent practices may result in WSDL documents with scarce relevant words, thus hindering the discoverability of their services. For instance, *Inappropriate or lacking comments*, *Ambiguous names* and *Whatever types* anti-patterns might reduce the number of relevant words within a WSDL document. On the other hand, frequent practices, namely *Ambiguous names*, *Low cohesive operations in the same port-type*, *Empty message*, *Redundant data model*, *Enclosed data model, Whatever types* and *Undercover fault information within standard messages* anti-patterns might add meaningless or unrelated words to WSDL documents.

Another discoverability obstacle stems from the introduction of redundant words. As syntactic registries compute how significant a word is to a service, according to the occurrences of the word within the WSDL file [Dong et al., 2004; Crasso et al., 2008b; Crasso et al., 2008a], redundant words may be considered as an attempt to influence the rank of Web Services returned by such a registry. This means that if a $word_w$ appears more times within $service_H$ than in $service_I$, the former service will be ranked first when using queries that contain $word_w$ [Dong et al., 2004; Crasso et al., 2008b; Crasso et al., 2008a]. The anti-patterns which might affect the importance of words are *Redundant port-types*, and *Redundant data models*.

Table 1: WSDL Discoverability anti-patterns.

Anti-pattern	Concerns	Symptoms	Manifests	Problems	Remedy
Inappropriate or lacking comments	Documentation	Occurs when (1) a WSDL document has no comments or (2) comments are too complex to be understood.	Evident (when 1) Not immediately apparent (when 2)	Conveys fewer, or none, words related to the functionality of the service.	Create concise comments and place it in the correct part of the WSDL document.
Ambiguous names [Blake and Nowlan, 2008]	Documentation	Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Not immediately apparent	Reduces the number of relevant words and introduces irrelevant ones.	Change ambiguous or meaningless names by explanatory names.
Redundant port-types	Design	Occurs when different port-types offer the same set of operations. Mostly because publishers re-define a port-type for each supported communication technology.	Evident	Influences the importance of words.	Summarize redundant port-types into a new port-type.
Low cohesive operations in the same port-type	Design	Occurs when port-types have weak cohesion, mostly because publishers include operations for monitoring the status of the service into the port-type that provides the offered functionalities.	Not immediately apparent	Introduces irrelevant words and influences their importance.	Draw operations having weak cohesion from their port-type and put them into a new port-type. Repeat until there are no port-types with poor levels of cohesion.
Empty messages	Design	Occurs when empty messages are used in operations that do not produce outputs nor receive inputs.	Evident	Introduces irrelevant words and influences their importance.	Remove empty messages and empty data-type definitions, if any.
Enclosed data model	Design	Occurs when the data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD ones.	Evident	Conveys fewer, or none, words related to the functionality of the service.	Move data-type definitions from WSDL documents to schema files.
Redundant data models	Representation	Occurs when many data-types to represent the same objects of the problem domain coexist into a WSDL document.	Evident	Introduces irrelevant words and influences their importance.	Summarize redundant data-types into a new data-type.
Whatever types [Pasley, 2006]	Representation	Occurs when a special data-type is used for representing any object of the problem domain.	Evident	Reduces the number of relevant words and introduces irrelevant ones.	Replace Whatever types with data-types that properly represent needed objects.
Undercover fault information within standard messages [Beaton et al., 2008]	Design	Occurs when output messages are used to notice about service errors.	Present in service implementation	Introduces irrelevant words and influences their importance.	Use WSDL fault messages for conveying error information.

We have studied practical solutions to every anti-pattern described in Table 1. Refactoring is the process of rewriting any software code to improve it in any way. According to the definition of refactoring, an anti-pattern remedy is a refactored design of WSDL documents. Here, refactorizations aim at improving service discoverability. Specifically, for remedying anti-patterns that reduce the number of available relevant words, the refactored solution often adds relevant words, which may be helpful in the search process. For example, the left side of Figure 3 depicts a Web Service description, whose offered operation lacks documentation, i.e., the WSDL document suffers from the *Inappropriate or lacking comments* anti-pattern. Following the remedy associated with this anti-pattern in Table 1, we have created concise comments and placed them inside the operation description, as it is showed within an ellipse in the right side of Figure 3. As a result, the refactored operation description conveys relevant words, such as "translate", "text", "English" and "German", which might ease discovering this service through syntactical registries.

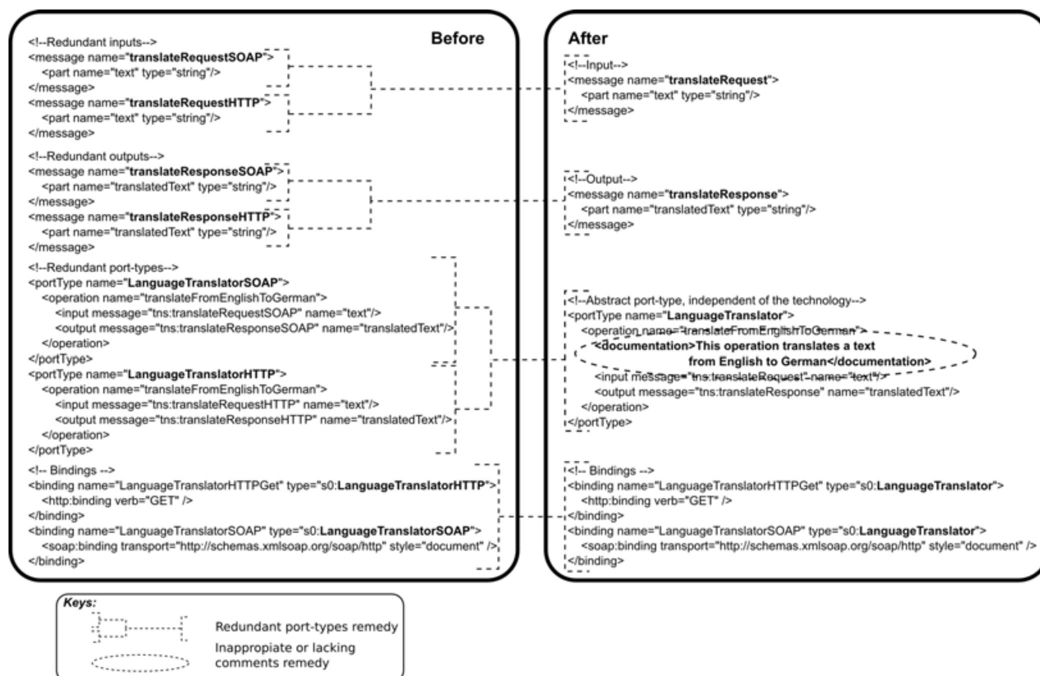

Figure 3: Remedying a WSDL document: before and after eradicating anti-patterns.

It is worth noting that the refactoring associated with the *Inappropriate or lacking comments* anti-pattern is compatible with agile software development methodologies, which suggest that programs should have concise comments.

The remedy for anti-patterns that introduce redundant words, comprises unifying redundant definitions into a unique and explanatory one. To clarify this, let us return to the service depicted on the left side of Figure 3, which suffers from the *Redundant port-types* anti-pattern as well. In this example, "LanguageTranslatorSOAP" and "LanguageTranslatorHTTP" port-types define the same set of operations and messages, twice. As described in Table 1, the solution is to summarize redundant port-types into a single one, while removing redundant words present in different port-type definitions. Graphically, dashed lines in Figure 3 represent the removal of redundant definitions from the WSDL document of the left side ("before"), and the creation of summarized ones in the WSDL document of the right side ("after").

Two, or more, anti-patterns can coexist in the same WSDL document, as is the case of the aforementioned example. This situation does not affect the proposed remedies, because they are independent of each other. This means that if a publisher sequentially uses the remedies related to "A", "B" and "C" anti-patterns to enhance a Web Service description, the resulting WSDL document will be free from "A", "B" and "C" anti-patterns.

For anti-patterns that introduce meaningless words, the proposed remedies aim to replace them with explanatory words. Explanatory words describe the semantics of the elements that they refer to. Semantically, a explanatory name should describe what its element represents, then meaningless names, such as “in0”, “arg1” or “foo”, should be avoided. Moreover, if there are two or more elements within a WSDL document standing for the same concept, these elements should be equally named. For instance, if an operation receives user’s details as input and another operation produces user’s details as output, their corresponding message parts should have the same name. Syntactically, on the other hand, the name of an operation should be in the form: <verb> “+” <noun>, because an operation is an action. For a message name, it should be a noun or a noun phrase, otherwise it may mean that a message conveys control information. Moreover, as syntactic registries rely on popular naming conventions, such as JavaBeans or Hungarian notations to split long names [Dong et al., 2004; Crasso et al., 2008b; Crasso et al., 2008a], if a name is composed by two or more words, the name should be written according to common notations. For example, the name “thisisthenameofanelement” should be rewritten as “thisIsTheNameOfAnElement” or “this_is_the_name_of_an_element”. Clearly, the latter names are easier to read than the original one. Another consideration is the length of a name. A name should neither be too short nor too long. A recommended length for the name of an WSDL element is between 3 and 15 characters [Blake and Nowlan, 2008].

The main goal of the proposed anti-pattern remedies is to improve the discoverability of Web Services, by refactoring WSDL documents. However, when a WSDL document not only defines the interface of an offered service, but also defines how it is used, it may be necessary to refactor the underlying service implementation as well. To clarify this, suppose an operation that suffers from *Undercover fault information within standard messages* anti-pattern. Clearly, though a publisher corrects the WSDL description of that operation, its implementation will still return control information as output. Therefore, it is essential to refactor the service implementation to eradicate the anti-pattern, and guarantee that the WSDL document is a reliable description of the offered interface. Accordingly, after applying the proposed remedies it is recommendable to verify whether the service functionality, which was working correctly before the refactorizations, still works as intended.

4 A survey of Discoverability Anti-patterns

We have analyzed 391 publicly available WSDL documents [Heß et al., 2004]. This data-set is divided into eleven categories:

- Business: consisting of 22 services. Their functionality range from creating bar-codes to calculating loan interest.
- Communication: 44 services related to different communication technologies such as sending e-mails, IMs and faxes as well as chatting. Besides, some services are focused on removing spam or accessing a directory.
- Converter: 45 services dedicated to convert measures, such as longitude, mass, weight and currency, from some unit to another.
- CountryInfo: 62 services in this category that includes searching for a Zip code, checking an address, obtaining the forecast for a region, getting directions and looking for near places, such as Churches or gas stations.
- Developers: 34 services oriented to developers needs. They allow creating proxy code for using other services, operate with date types, search things in Amazon, validate license numbers or hash a password.
- Finder: 44 services to perform different kinds of search. For instance, they permit searching Internet for a Web site, looking up a word in a dictionary or a chemical element in the periodic table.
- Games: 9 services for playing chess, lottery and other board games.
- Mathematics: 10 services for executing complex mathematical operations.

- Money: 54 services for converting currencies, calculating mortgage, operating in the stock market and performing payments.

- News: services for consulting different news sources. There are 30 services in this category.

- Web: 37 services useful to develop Web applications. Services in this category provide login capabilities, rendering information in HTML or searching information in a database.

Initially, we manually revised 130 files of the data-set and documented the catalog of WSDL discoverability anti-patterns of Section 3. To have an assessment of how common these bad practices are, we analyzed the whole corpus of WSDL files.

Accordingly, each bar of Figure 4 depicts the number of WSDL files that suffer from an anti-pattern (not the number of anti-pattern symptom occurrences). The reason to present the results in this way, is that the symptoms of some anti-patterns may occur more than once in a WSDL document (*Ambiguous names*, *Empty messages*, *Redundant data models*, *Whatever types*, *Undercover fault information within standard messages*), but the symptoms of other anti-patterns may occur only once in a WSDL file (*Inappropriate or lacking comments*, *Redundant port-types*, *Low cohesive operations in the same port-type*, *Enclosed data model*). Therefore, assessing the number of service descriptions that suffer from each anti-pattern, provides evidence of how important each anti-pattern is for the employed data-set.

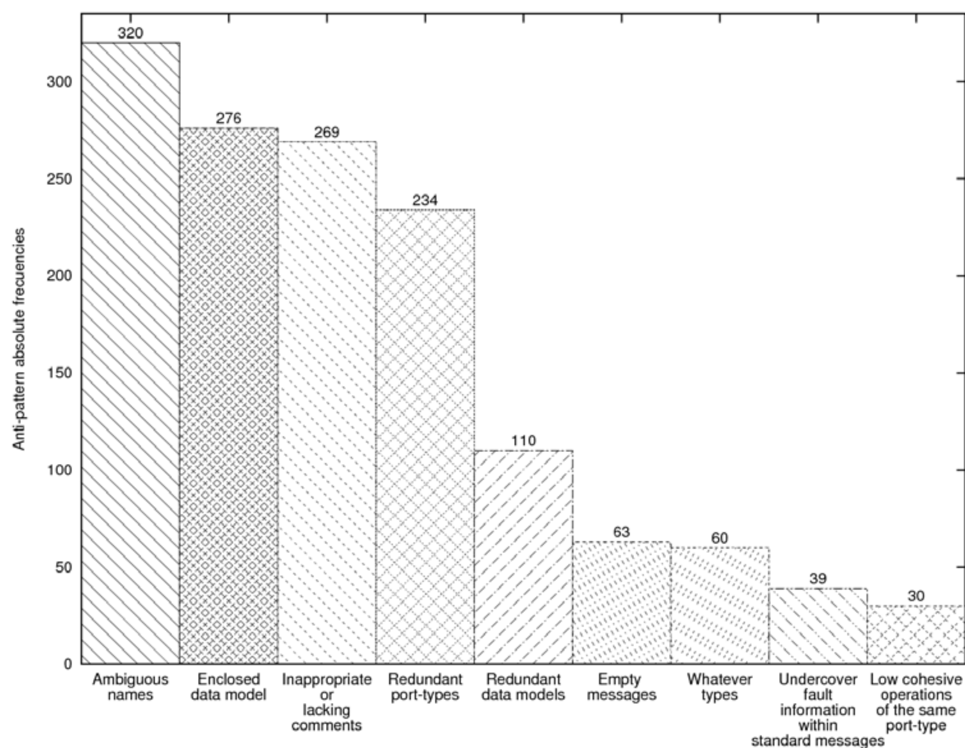

Figure 4: Anti-pattern occurrences within 391 Web Services.

The results show that some anti-patterns affect more WSDL documents than others, but even the least frequent anti-pattern occurs in 30 WSDL documents. Notably, in spite of the intuitive importance of good naming and commenting practices, 82% and 69% of the documents suffer from *Ambiguous names* and *Inappropriate or lacking comments* anti-patterns, respectively. Additionally, the *Enclosed data model* anti-pattern, which not only hinders discovery, but also constrains data-model re-utilization, occurs in 70% of the documents. Similarly,

notwithstanding the fact that port-types are meant to define the service functionality independently of any technological aspect, the *Redundant port-types* anti-pattern affects 60% of the documents. On the other hand, the anti-pattern that appears less often within this data-set is the *Low cohesive operations in the same port-type* anti-pattern: 7.6%. Similarly, the proportion of documents that suffers from *Undercover fault information within standard messages* anti-pattern is 10%.

A collateral finding of this study is that detecting an anti-pattern is strongly related to the way it manifests itself. As shown in Table 1, anti-patterns can be classified as being: "Evident", "Not immediately apparent" and "Present in service implementation". First, "Evident" anti-patterns provide visible signs in WSDL documents. For this type of anti-patterns, it is easy to define a detection criterion based on the syntax of a WSDL document. A clear case of this is *Enclosed data model* anti-pattern, since it can be deterministically detected by applying the following rule: "if at least a type is defined in a WSDL document, then the anti-pattern occurs". Second, "Not immediately apparent" anti-patterns cannot be detected by analyzing the syntax of a WSDL document, because they are strongly related to the intended meaning of an element. A detection criterion for this kind of anti-patterns is more complex and involves questions like "Is the name of this message part ambiguous?" or "Is the documentation of this operation clear enough?". Third, sometimes "Present in a service implementation" anti-patterns have no footprint in a WSDL document, and they can only be detected by invoking its implementation. If a message includes a part to inform whether an error has occurred, or this situation is explicitly explained in the comments of the operation, the anti-pattern can be detected. Instead, if an output message part is called "parameter" and it exchanges a *Whatever type*, it might be used to inform an error, which would be a case of *Undercover fault information within standard messages* anti-pattern but it is impossible to know for certain. Here, the *Undercover fault information within standard messages* anti-pattern cannot be detected, unless the service implementation fires an error during a request. For the study presented in this section we consider the anti-pattern only if it can be detected in the WSDL document.

5 A case study of applying Discoverability Anti-patterns Remedies

In order to illustrate how to apply the proposed anti-pattern remedies, we selected a WSDL document[4] that suffers from several anti-patterns from the Web Service corpus described above. Throughout this section we show which anti-patterns affect the WSDL document and how to remove them. Finally, we discuss the resulting WSDL document and compare it with its original version.

The selected WSDL document describes a service for converting a force expressed in some unit to another unit, as shown in Figure 5. Despite having three port-types, the selected WSDL document defines only one operation, named "ChangeForceUnit". The reason to define three different port-types for the same operation is that each port-type corresponds to a single transport protocol, such as SOAP, HttpGet or HttpPost. Since port-types should be protocol-independent, this WSDL document suffers from *Redundant port-type* anti-pattern. Another characteristic of the WSDL document is that the operation has no comments. Therefore, the *Inappropriate or lacking comments* anti-pattern affects the WSDL document. Analyzing the input and output messages of the operation, the names related with some parts are not explanatory of the information that they convey. For instance, the name for both input and output SOAP message parts is "parameters". Clearly, every message part is a parameter; however, the name should represent the parameter semantics. Furthermore, in both HttpGet and HttpPost, the output message part is named as "Body", which refers to an HTML body tag usually associated with the Http protocol. This is a symptom of the *Ambiguous names* anti-pattern.

Regarding the data-model there are two aspects: first, there exists duplicated data definitions, and second, the data model is enclosed in the WSDL document. The first problem is related to the definition of the type "s0:double", which is defined as a primitive type "s:double", which is also accessible from the WSDL document. This manifests the *Redundant data models* anti-pattern. On the other hand, the *Enclosed data model* anti-pattern is present in this WSDL document because the data-model definition is inside the WSDL document. As a result, it is impossible to reuse the data model for another service that also uses force magnitude as input or output.

---

4 WSDL document, http://www.webservicex.net/ConvertForec.asmx?WSDL

For improving the WSDL document, we recommend to remove one anti-pattern at a time. First, it is necessary to choose the order in which the anti-patterns will be removed. Even though the remedies can be applied independently, if the *Enclosed data model* anti-pattern is removed first, the size of the WSDL document will be reduced, making it easier to handle. Second, removing redundant information (the *Redundant port-types* and *Redundant data models* anti-patterns) also removes redundant instances of other anti-patterns. For instance, if a port-type has no documentation and it is repeated in the WSDL document, all the redundant port-types will have no documentation. Lastly, the remaining anti-patterns can be solved in any order.

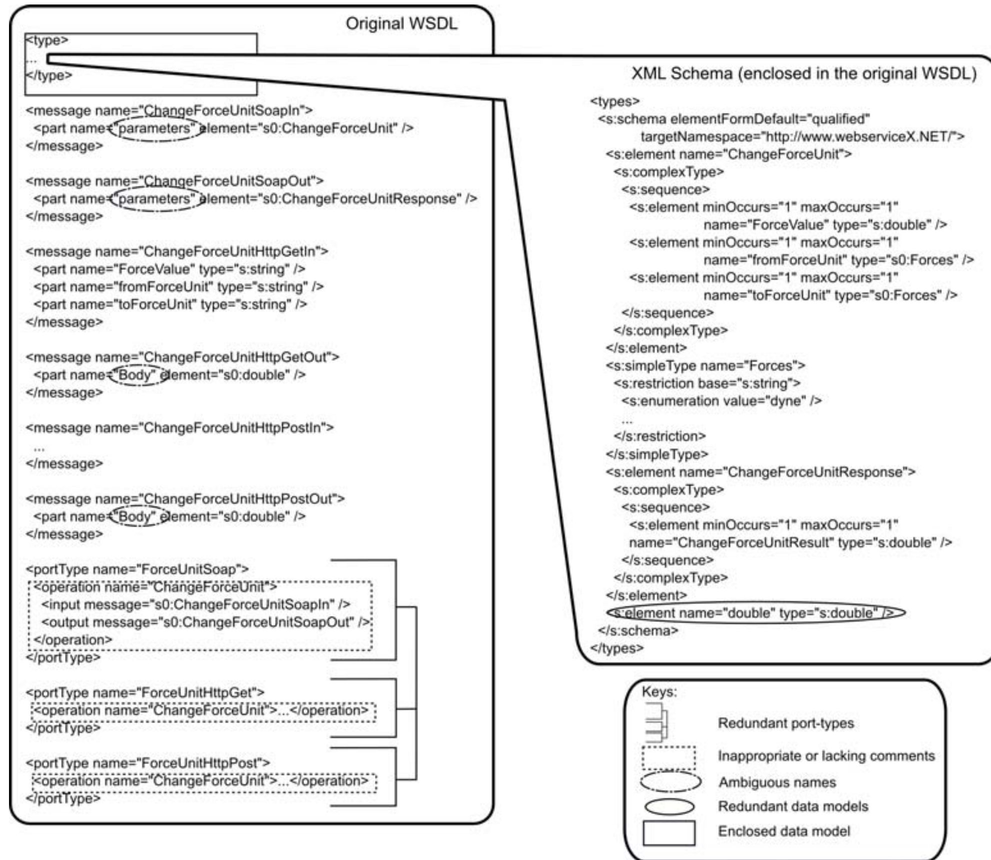

Figure 5: Original WSDL document.

Therefore, to start fixing the case study WSDL document, we removed the *Enclosed data models* anti-pattern by moving the data model into a separated XSD file and importing it from the WSDL document. As a result, we divided the WSDL document into two files for the sake of describing the service, one for describing the service itself and the other one for defining the data model used by the service. Then, the *Redundant port-type* anti-pattern was removed by dropping all redundant port-types, but one. In addition, the messages that were no longer referenced by any port-type were also removed. Besides, any transport protocol reference, such as SOAP, HttpGet or HttpPost, was removed from the port-type name and from the message names. We arbitrarily decided to keep the "ForceUnitHttpGet" port-type, but the refactoring could be done keeping "ForceUnitSoap" or "ForceUnitHttpPost" port-type as well.

The other problem related with redundant code is the *Redundant data model* anti-pattern. Through an analysis of the WSDL document, we can notice that invoking the service requires to transport two data types: strings to carry the force unit and doubles to express the measure of the force. Because there is only a very limited subset of strings

that actually represent a force unit, such as, dyne, gramforce, newtons, we decided to define a restrictive type that accepts only those strings. On the contrary, there is no constrain on the value of the double that represents the force measure. Consequently, we decided that using the primitive type double would be enough. Clearly, this is not the only possible solution. For instance, another solution could be to mix the force measure and the force unit in a single data type that expresses a force.

As a result of removing redundant port-types, the only remaining ambiguous name in the WSDL document is "Body", which stands for the only output message part. Following [Blake and Nowlan, 2008], this name should be replaced by a more descriptive one. This message part conveys the value of the converted force. Since the input message part "ForceValue" stands for the value of the original force, we think that "ForceValue" may be an accurate name for the output message part. Finally, as many WSDL documents, this one has no documentation [Fan and Kambhampati, 2005]. Therefore, we added a <documentation> element into the description of the offered operation, which tells that this operation converts force measures.

The resulting WSDL document after applying the anti-pattern remedies is shown in the Figure 6. An important difference between the original version of the WSDL document and the improved one is that the latter is shorter than the original version. However, the reduction of a WSDL document is not always caused by removing the anti-patterns. For instance, if the original WSDL document lacks documentation, the resulting WSDL document will be longer. However, the number of unrepresentative and ambiguous words, such as SOAP, HttpPost or HttpGet, in the improved WSDL document are fewer than in the original one. On the other hand, the number of relevant words, i.e., force, convert, change, in the improved WSDL document is higher than in the original one. Finally, the improved WSDL document has comments that the service consumer can read to understand what the service does, and how it should be invoked.

```
<?xml version="1.0"?>
<s:schema xmlns:s="http://www.w3.org/2001/XMLSchema">
<s:simpleType name="Forces">
 <s:restriction base="s:string">
   <s:enumeration value="dyne" />
   <s:enumeration value="gramforce" />
   <s:enumeration value="poundals" />
   <s:enumeration value="newtons" />
   <s:enumeration value="pounds" />
   <s:enumeration value="kilopondkgmforce" />
   <s:enumeration value="Kip" />
 </s:restriction>
</s:simpleType>
</s:schema>
```

```
<message name="ChangeForceUnitIn">
  <part name="ForceValue" type="s:double" />
  <part name="fromForceUnit" type="s0:Force" />
  <part name="toForceUnit" type="s0:Force" />
</message>

<message name="ChangeForceUnitOut">
  <part name="ForceValue" element="s:double" />
</message>

<portType name="ChangeForceUnit">
  <operation name="ChangeForceUnit">
     <documentation>This service converts a force measure
      in a force unit to the same measure
      in another force unit</documentation>
   <input message="s0:ChangeForceUnitIn" />
   <output message="s0:ChangeForceUnitOut" />
  </operation>
</portType>
```

Figure 6: Improved WSDL document and XSD file

6 Experimental Evaluation

In order to determine the impact of the presented anti-patterns, we conducted two different experiments. The first one for measuring the impact of the anti-patterns on a syntactic registry. The second experiment was designed to analyze whether or not WSDL documents without anti-patterns are more readable for humans than WSDL documents with anti-patterns.

6.1 Anti-patterns effect on syntactic registries

We have assessed the retrieval effectiveness of a syntactic registry using two versions of the data-set described in Section 4: one comprising the original WSDL documents, and another one comprising corrected WSDL documents. We built the improved version of the data-set by removing all the found anti-patterns. While doing this, we measured the time required to improve a WSDL document that was 15 minutes on average. Our goal is to have an assessment of the discovery effectiveness improvements introduced by removing the anti-patterns of Section 3 from real world services. To do this, two instances of the same syntactical service registry, i.e., it operates by comparing words extracted from WSDL documents with words present at queries, were deployed and started. Then, one registry was supplied with the original data-set, whereas the other registry was fed with the enhanced data-set. It is beyond the scope of this paper to formally present the characteristics of the employed registry, however it is important to mention that this registry, called WSQBE, returns a ranked list of WSDL documents relevant to a given query, being the document at the top of the rank the most relevant service to the query, and so on. A complete study of WSQBE can be found in [Crasso et al., 2008b; Crasso et al., 2008a]. Once the two data-sets were published, the same 30 queries, which are described in [Crasso et al., 2008a], were used to discover services through both registries. Finally, metrics were taken to evaluate the impact of removing WSDL anti-patterns.

We decided to employ the measures summarized in Table 2, because they not only consider how well an engine performs in finding relevant documents but they also take into account the position of each relevant retrieved service within the result list. This fact makes these measures especially suitable for comparing registries that arrange retrieved services in a regular formation, as WSQBE does. As some of these measures require knowing exactly the set of all services in the collection relevant to a given query, we have exhaustively analyzed the corpus of WSDL documents to determine the relevant services for each query. To do this, a developer judged whether or not the operations of a retrieved service fulfilled the expectations previously specified in each query.

Table 2: Summary of employed information retrieval measures.

Measure:	Computes:	Formula:
Recall-at-*n*	The proportion of retrieved relevant documents (*RetRel*) within a result list of size=*n*, where *R* represents the number of services relevant to a query.	$\frac{RetRel_n}{R}$
Normalized Recall	The position (r_i) of each relevant document (i^{th}) in the result list, where *N* is the size of the data-set.	$1-\frac{\sum_{i=1}^{R} r_i - \sum_{i=1}^{R} i}{R(N-R)}$
Precision-at-*n*	The precision at different cut-off points of the result list.	$\frac{RetRel_n}{n}$
R-precision	The precision at the *R* position in the ranking.	$\frac{RetRel_R}{R}$

We have calculated the aforementioned measures for each query and then averaged the results over the 30 queries. Figure 7 shows the average results of each measure. In order to enable comparisons, we arranged the results in groups of two bars, in which each group is associated with an employed measure. From left to right, the first bar within each group depicts the achieved results when using the original version of the data-set, while the second bar represents the results when using the improved version.

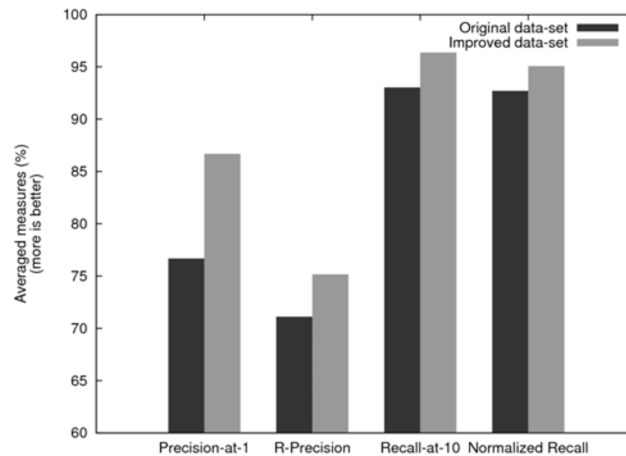

Figure 7: Average measure results and how anti-patterns affect matching sensitivity.

Figure 7 shows that all measures were better after removing anti-patterns from the data-set. The biggest gain takes place in the results associated with the Precision-at-*1* measure, in which the experiments with the improved WSDL documents surpassed by 10% their counterpart, on average. Having a higher Precision-at-*1* means that the registry performs better in retrieving a relevant service at the top of the result list. These results stem from the fact that the original WSDL documents usually contain redundant, meaningless and nonspecific terms. Specifically, the original WSDL documents have 3368 unique terms, but after applying the proposed anti-pattern solutions they only have 2555 unique terms. Indeed, as the proposed anti-pattern solutions remove meaningless nonspecific terms and add explanatory names, the refactored WSDL documents have fewer terms, but they are more explanatory of the service functionality. The gain of 4% in *R*-precision results, has provided more evidence of the improvements in the retrieval of relevant services before non-relevant ones, when removing all WSDL anti-patterns from the data-set. *R*-precision computes the precision at the Rth position in the ranking, where *R* is the number of services in the data-set relevant to a given query. This measure is a special case of Precision-at-*n,* when $n=R$.

The results have empirically shown that, when using the improved data-set, WSQBE was more effective in retrieving more relevant services as well. Recall-at-10 and Normalized Recall results confirm this fact. Recall is a measure of how well a recommender system performs in finding relevant documents, services in this case, by finding out how many relevant services are included in the result list. Using the improved data-set, WSQBE achieved Recall-at-*10* of 96.36%, whereas using the original documents it achieved 93.02%. Normalized Recall, on the other hand, takes into account Recall and the position of each relevant retrieved service within the result list. WSQBE achieved a Normalized Recall of 95.05% and 92.69% for the improved data-set and the original one, respectively. Therefore, when removing discoverability anti-patterns, the employed registry not only retrieved more relevant services, but also ranked them first in the result list. These results are significant because of users' tendencies to select higher ranked search results, even a small improvement in a rank has a great impact on discoverability [Agichtein et al., 2006]. For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the second ranked one is, at most, 60% [Agichtein et al., 2006]. This further strengthens the importance of removing discoverability anti-patterns from WSDL documents, in light of the significant Precision-at-*1* improvements observed in these experiments.

6.2 Anti-patterns effect on WSDL documents intelligibility

As it was showed in the previous section, not all the anti-patterns affect the same fraction of the WSDL documents corpus. As a consequence, the anti-patterns that occur in fewer WSDL documents than the commonest ones, might not impact on syntactic registries in the same degree. However, those anti-patterns might affect how a human discovers, selects and understands a particular service. The uncommonest anti-patterns that seem not to be relevant for syntactic registries are *Low cohesive operations of the same port-type*, *Undercover fault information within*

standard messages, *Whatever types* and *Redundant data models*. To analyze the implications of those anti-patterns, we conducted a survey to determine whether or not the intelligibility of a WSDL document that suffers from the uncommonest anti-patterns improves after it has been refactored according to our guide. We asked software engineering students and professionals about the intelligibility of two versions of the same WSDL document: the original WSDL document in the Web Service corpus, and the improved WSDL document. Concretely, the participants were given one of two WSDL documents and a questionnaire designed to analyze the implications of the anti-patterns that less often occur in the analyzed data-set.

We selected two WSDL documents that have the aforementioned 4 anti-patterns from the data-set used in the first experiment. Then, for each service we generated an improved WSDL document by removing the anti-pattern occurrences. The participants were asked several questions about the semantic information of the service contained in the original version of each WSDL document. Afterward, they were asked about the same information, but this time contained in the improved WSDL documents. Those questionnaires were design to help the participants with understanding the services. Finally, a third questionnaire was given to the participants to ask about both services in particular which of them they would select and why. The presented data was collected from this questionnaire. The survey was conducted as homework, with 26 participants who were taking a SOC course[5] at the UNICEN. The group of participants was integrated by last year software engineering students and practicing software engineers. It is worth noting that the participants did not know the catalog of discoverability anti-patterns proposed in this paper or its related works, until the survey was finished.

First, we asked the participants if they believed that the low cohesive operation should be removed from the port-type. 92% of the participants responded that they agree on that. Second, we asked how the error information should be returned. 20% of the participants answered that the error information should be returned within the output message, while 80% affirmed that the error information should be returned in a fault message. Finally, when they were asked if the data-types should be reused or if different data-types should be created for each operation, even if that means creating redundant data-types, 80% of the participants suggested that WSDL documents should not have redundant data-type definitions.

In order to evaluate which service definition the participants preferred, we asked which service they would select and why. 84% of the participants stated that they would select the improved version of the WSDL document, while 8% of them preferred the original version. The other 8% of the participants did not express a preference. Each participant was asked about the reasons for their selection. We detected four main reasons given by the participants who would select the improved WSDL document:

1. data-types are better represented and easier to understand than in the original WSDL document (16 participants),

2. the improved WSDL document is more consise than the original (13 participants),

3. port-types contain only semantically related operations (8 participants),

4. error information is better handled in the improved WSDL document (6 participants).

Those reasons are directly related to the uncommonest anti-patterns that we removed from the original version of the WSDL document. In particular, the first reason is related to the *Redundant data models* anti-pattern and the *Enclosed data model* anti-pattern because the improved data model design is a direct result of applying those anti-pattern remedies to the original WSDL document. From the number of participants who gave this reason, it seems that a well-designed data model is considered very important for potential Web Service consumers. In addition, the second reason pointed out that Web Service consumers usually prefer services defined by short WSDL documents, with concise comments. Those qualities are related to the remedies of four anti-patterns: *Inappropriate or lacking comments*, *Redundant port-types*, *Redundant data models* and *Enclosed data model*. The other two reasons given by the participants are related to remedying the *Low cohesive operations in the same port-type* and *Undercover fault information within standard messages* anti-patterns, respectively.

---

5 SOC course at the UNICEN, http://www.exa.unicen.edu.ar/~cmateos/cos

7 Conclusions and Future Work

This paper presented a novel catalog of nine WSDL discoverability anti-patterns. We have analyzed a corpus of 391 real Web Services, and found nine frequent practices that may degrade the retrieval effectiveness of syntactic registries, while hindering users' ability to understand third-party services. We have found that 82% of the documents from the data-set contain, at least, one anti-pattern, while each anti-pattern affects 40% of the documents in the data-set, on average.

This paper proposes a reproducible solution to each identified bad practice. We have empirically validated that by applying the proposed solutions, the retrieval performance of a syntactic registry improves by: 10% for Precision-at-1, 4.05% for *R*-Precision, 3.34% for Recall-at-10 and 2.36% for Normalized Recall. It is worth noting that retrieval effectiveness improvements can be data-set specific. Therefore, these results cannot be generalized to other data-sets of WSDL documents. Nevertheless, as our approach relies on removing meaningless or unnecessary information and incorporating self-descriptive names and comments, it is reasonable to expect at least a small performance improvement when eradicating WSDL anti-patterns.

We also performed a survey to collect subjective opinions from software engineering students and practicing engineers about the intelligibility of several WSDL documents. The results of the survey suggest that WSDL documents should be improved according to the proposed solutions to increase service chances of being understood and, in turn, outsourced.

The experiment presented in this paper also shows that to enhance a WSDL document and to verify that its semantics have not changed, an experienced service-oriented application developer requires 15 minutes, on average. Therefore, we believe that manually enhancing WSDL documents should be incorporated as a development task because 15 minutes is a reasonable time investment with a favorable outcome of making services easier to understand and discover by potential consumers.

Future work related to the identified anti-patterns is planned in two directions. The first direction is mainly experimental, and it is aimed to collect evidence of the individual impact of each identified anti-pattern. Preliminary results point out that removing the *Inappropriate or lacking comments* anti-pattern contributed to a better effectiveness of the employed service registry than removing the other anti-patterns. Instead, the lowest impact is being caused by removing the *Empty message anti-pattern*. Additionally, we are evaluating the impact of the anti-patterns using different service registries. Preliminary results related to removing anti-patterns surpass those achieved by using the original data-set with all the registries employed. This fact provides empirical evidence that suggests that the improvements are explained by the removal of discoverability anti-patterns rather than the incidence of the underlying discovery mechanism.

We are also researching on heuristics for automatically detecting poor practices in Web Service descriptions. We aim to assist developers in making more representative descriptions, by automatically identifying the poor practices and suggesting suitable refactorizations.

The other research direction concerns with Service-Oriented Grids, in the same fashion as we did for SOC. A Service-Oriented Grid is a heterogeneous network with a middleware for high throughput computing, in which all the functions of the middleware, such as security, storage or scheduling, are offered as Web Services [Grimshaw et al., 2009], thus these must be discovered to be exploited. Web Services are used to provide support for Grid infrastructures [Atkinson et al., 2005], as well as to supply consumers with specific functionalities. Clearly, service discovery is a crucial task for this approach to succeed. Therefore, we believe that remedying frequent discoverability problems may allow developers to easily "plug-in" applications into Grids [Mateos et al., 2008a; Mateos et al., 2009; Mateos et al., 2008b].

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper. We also thank Cristian Mateos for helping us to perform the survey. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

Agichtein, E., Brill, E., Dumais, S., and Ragno, R. (2006). Learning user interaction models for predicting web search result preferences. In SIGIR'06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pages 3–10, New York, NY, USA. ACM.

Al-Masri, E. and Mahmoud, Q. (2008). Discovering Web Services in search engines. IEEE Internet Computing, 12(3):74–77.

Atkinson, M., DeRoure, D., Dunlop, A., Fox, G., Henderson, P., Hey, T., Paton, N., Newhouse, S., Parastatidis, S., Trefethen, A., Watson, P., and Webber, J. (2005). Web Service Grids: An Evolutionary Approach. Concurrency and Computation: Practice and Experience, 17(2-4):377–389.

Beaton, J., Jeong, S. Y., Xie, Y., Jack, J., and Myers, B. A. (2008). Usability challenges for enterprise Service-Oriented Architecture APIs. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 193–196.

Bichler, M. and Lin, K.-J. (2006). Service-Oriented Computing. Computer, 39(3):99–101.

Blake, M. B. and Nowlan, M. F. (2008). Taming Web Services from the wild. IEEE Internet Computing, 12(5):62–69.

Crasso, M., Mateos, C., Zunino, A., and Campo, M. (2010). Empirically assessing the impact of dependency injection on the development of Web Service applications. Journal of web Engineering, 9(1):66–94.

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

Crasso, M., Zunino, A., and Campo, M. (2008b). Query by example for Web Services. In SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, pages 2376–2380, New York, NY, USA. ACM.

de Souza, C. R. B., Redmiles, D., Cheng, L.-T., Millen, D., and Patterson, J. (2004). How a good software practice thwarts collaboration: the multiple roles of apis in software development. SIGSOFT Software Engineering Notes, 29(6):221–230.

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., and Zhang, J. (2004). Similarity search for Web Services. In Proceedings of the 13th International Conference on VLDB, pages 372–383, Toronto, Canada. Morgan Kaufmann.

Erickson, J. and Siau, K. (2008). Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. Journal of Database Management, 19(3):42–54.

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

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

Grimshaw, A., Morgan, M., Merrill, D., Kishimoto, H., Savva, A., Snelling, D., Smith, C., and Berry, D. (2009). An open Grid Services architecture primer. Computer, 42(2):27–34.

Henning, M. (2009). API design matters. Communications of the ACM, 52(5):46–56.

Heß, A., Johnston, E., and Kushmerick, N. (2004). ASSAM: A tool for semi-automatically annotating semantic Web Services. In 3rd ISWC, volume 3298 of LNCS, pages 320–334.

Hummel, O., Janjic, W., and Atkinson, C. (2008). Code conjurer: Pulling reusable software out of thin air. IEEE Software, 25(5):45–52.

Mateos, C., Zunino, A., and Campo, M. (2008a). JGRIM: An approach for easy gridification of applications. Future Generation Computer Systems, 24(2):99–118.

Mateos, C., Zunino, A., and Campo, M. (2008b). A survey on approaches to gridification. Software: Practice and Experience, 38(5):523–556.

Mateos, C., Zunino, A., and Campo, M. (2009). Grid-enabling applications with JGRIM. International Journal of Grid and High Performance Computing, 1(3):52–72.

McConnell, S. (2006). Software Estimation: Demystifying the Black Art. Microsoft Corporation, Redmond, USA.

McCool, R. (2006). Rethinking the Semantic Web, part II. IEEE Internet Computing, 10(1):96, 93–95.

McIlraith, S., Son, T., and Zeng, H. (2001). Semantic Web Services. Intelligent Systems, IEEE, 16(2):46–53.

Nakamura, S., Konishi, S., Jatowt, A., Ohshima, H., Kondo, H., Tezuka, T., Oyama, S., and Tanaka, K. (2007). Trustworthiness analysis of web search results. In Research and Advanced Technology for Digital Libraries (LNCS), pages 38–49.

Paolucci, M. and Sycara, K. (2003). Autonomous semantic Web Services. IEEE Internet Computing, 7(5):34–41.

Pasley, J. (2006). Avoid XML Schema wildcards for Web Service interfaces. IEEE Internet Computing, 10(3):72–79.

Pendharkar, P. C. and Rodger, J. A. (2002). An empirical study of factors impacting the size of object-oriented component code documentation. In SIGDOC ’02: Proceedings of the 20th annual international conference on Computer documentation, pages 152–156, New York, NY, USA. ACM Press.

Sabou, M. and Pan, J. (2007). Towards semantically enhanced Web Service repositories. Web Semantics: Science, Services and Agents on the World Wide Web, 5(2):142–150.

Song, H., Cheng, D., Messer, A., and Kalasapur, S. (2007). Web Service discovery using general-purpose search engines. In IEEE International Conference on Web Services (ICWS), pages 265–271.

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