Discoverability anti-patterns: frequent ways of making undiscoverable Web Service descriptions

Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino and Marcelo Campo

ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682. Fax.: +54 (2293) 439681

Abstract. The ever increasing number of publicly available Web Services makes standard-compliant service registries one of the most essential tools for service-oriented application developers. Previous works have showed that the retrieval effectiveness of this kind of registries strongly depends on the descriptiveness of published service descriptions. 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.

Web Services; Web Service discoverability anti-patterns.

1 Introduction

An evolutionary process currently taking place in the software industry is the shift 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 search software pieces with publicly available interfaces, called *services*, within specialized registries to form their applications [1,2]. 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 at software development in heterogeneous distributed environments [2] 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 [3].

Currently, the SOC paradigm is mostly implemented by using Web Services technologies [2]. A Web Service is a software system that can be discovered and invoked through standard Web protocols. The growth of the Internet has popularized large repositories of Web Services [1,4]. Unfortunately, finding proper services is very challenging, mainly because, unlike traditional software libraries, Web Service repositories rely on little meta-data to support service discovery [5] and the ever increasing number of published Web Services [4]. In the end, these challenges hinder the adoption of the SOC paradigm [6].

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 of supplying a registry with the information associated with providers and technical descriptions of the functionalities of their services in Web Service Description Language (WSDL)[2]. WSDL is an XML-based language for describing a service intended functionality by means of an interface with methods and arguments, in object-oriented terminology, and documentation in the form of textual comments. On the other hand, discoverers may look up third-party services by performing keyword-based searches.

Mostly because of the inconsistency between keywords in interfaces of publicly available services and queries [7,8], efficiently finding proper services through implementations of UDDI is similar to finding a needle in a haystack [4]. Nowadays, three main directions have been proposed to cope with this problem. As service descriptions usually reside on Web servers, one direction proposes to exploit the capabilities of Web search engines (e.g., Google) to crawl and index Web servers content [9,10]. Although this approach is transparent to publishers, several studies have empirically showed that the precision of Web searchers when looking for known services does not significantly improve, even when proper comments had been introduced in the indexed WSDL documents [9]. Accordingly, Web Service discoverers experience the same problems as ordinary users of Web searchers [11].

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 [8,12,13,14,15]. 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. By assuming that all the aspects surrounding services are precisely described, it is expected that finding them would 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 [16]. Obviously, syntactic approaches cannot replace the need for semantic machine-interpretable descriptions in the context of automatic applications, which discover services without human intervention [7,8]. However, several studies [12,14,15] have empirically shown that syntactic approaches can effectively facilitate 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 [16].

[1] UDDI `http://uddi.xml.org/`

[2] WSDL `http://www.w3.org/TR/wsdl`

In the rest of the paper we will focus only on approaches where service publishers and discoverers are human beings, instead of computer programs as most semantics-based approaches do.

Unfortunately, the aforementioned promising results [12,14,15] can not be generalized and may vary with different data-sets, though their corresponding efforts have been rigorously evaluated. In fact, as the underpinnings of syntactic approaches to service discovery lie in the descriptiveness of service specifications, WSDL documents without any proper comments or any representative keywords may deteriorate their retrieval effectiveness. In spite of the intuitive implications of the use of non representative 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 [17,18]. However, as far as we known, software practitioners lack of empirical evidence showing whether detected frequent programming conventions and design practices occur in Web Services development or not, and whether any practices affect the discoverability of services.

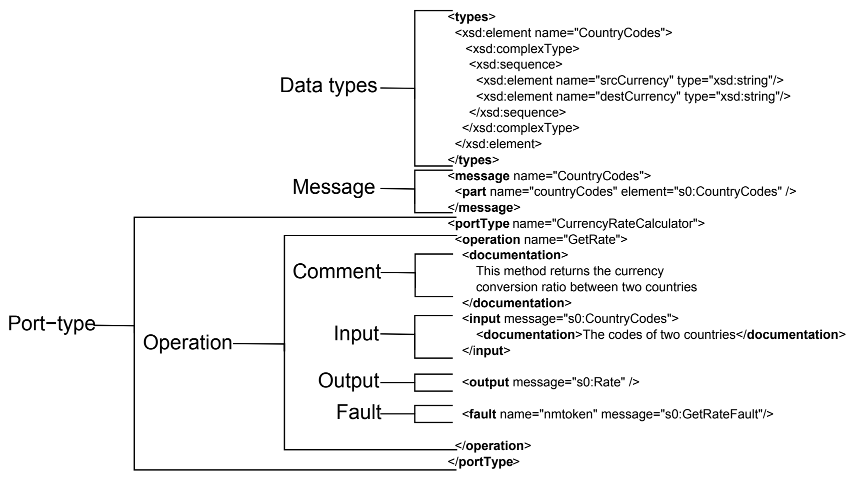
In order to assist publishers in the creation of services that can be easily discovered through syntactical and standard-compliant approaches, we have studied common practices found in a corpus of real world Web Service descriptions that, paradoxically, may attempt against service discoverability. This paper describes how to solve these frequent problems in a novel catalog of Web Service discoverability anti-patterns, which subsumes previous work on making improved WSDL documents [8,19,20] and points out even more bad practices. 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. To sum up, the main contributions of this paper are:

– 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
– 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 and syntactic approaches for discovering services. Section 3 discusses related works. Then, Sect. 4 presents WSDL anti-patterns. Later, in Sect. 5 and Sect. 6 we survey the presence of anti-patterns in real Web Services and show the implications of solving them when using a syntactic registry, respectively. Lastly, Sect. 7 concludes the paper.

2 Background

The WSDL language allows developers to describe two main properties of a service: its functionality and how to invoke it. Discoverers use the functional descriptions to match third-party services against their needs and, in turn, they take under consideration the technological details for invoking a selected service. A WSDL document describes the functionality of a service 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 depicts a concrete WSDL document.



```
<types>
 <xsd:element name="CountryCodes">
   <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="srcCurrency" type="xsd:string"/>
      <xsd:element name="destCurrency" type="xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
</types>
<message name="CountryCodes">
 <part name="countryCodes" element="s0:CountryCodes" />
</message>
<portType name="CurrencyRateCalculator">
 <operation name="GetRate">
   <documentation>
      This method returns the currency
      conversion ratio between two countries
   </documentation>
   <input message="s0:CountryCodes">
      <documentation>The codes of two countries</documentation>
   </input>

   <output message="s0:Rate" />

   <fault name="nmtoken" message="s0:GetRateFault"/>

 </operation>
</portType>
```

Fig. 1. Example of Web Service description using WSDL.

Messages comprise parts that transport structured data. Exchanged information is arranged according to specific data-type definitions using the XML Schema Definition (XSD)[3], which is a language to express the structure of XML content. The WSDL document depicted in Fig. 1 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 other WSDL documents afterward.

In conclusion, the WSDL language 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 in the query. Afterward, a rank of services is built according to the similarity between the query and the extracted words. Then, besides the well known goodness of documenting software and using representative names [21], the retrieval effectiveness of syntactic registries, heavily depends on how effective for discovery the WSDL documents and queries are [12,13,14,15]. Therefore, the quality of WSDL descriptions is crucial for syntactical approaches to Web Service discovery.

[3] XSD `http://www.w3.org/XML/Schema`

3 Related Work

As far as we know, there are three works that address the problem associated with the quality of WSDL documents from the perspective of discovery through syntactic service registries. In [19] 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., “Pay-Order” or “DatabaseRecord”. Clearly, this kind of definition does not allow discoverers to exactly infer what the input/response of a service is like. Although using wild-cards creates vague interface contracts, one detected reason for using them is 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 correctly invoke and process the operations defined in its new version [19].

In [20] the authors present many difficulties 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 [20]. 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 [8] 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 XSD wild-cards [19], control parameters [20] and naming tendencies [8], but also supplies each problem with a practical solution.

4 Web Service Discoverability Anti-Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. Anti-patterns extend the notion of patterns to express obvious, wrong, solutions to recurring problems that have been supplied with refactored solutions that are clearly documented, proved and repeatable. Building a catalog of design anti-patterns means to represent design errors, which are known to occur frequently in a large number of applications, to enable their detection and solution.

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. [22]. Each of the bad practices has been studied to provide a sound and practical solution. Table 1 shows a comprehensive list of WSDL discoverability anti-patterns, which presents the next information separated in columns:

- Anti-pattern: a descriptive name for the anti-pattern.
- Concern: 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 are 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.
- Manifest: 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 semantic, in order 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 Sect. 6.
- Problems: an enumeration of the problems the anti-pattern can cause to a syntactic registry.
- Remedies: a list of steps to follow in order to resolve the problems that can be caused by the anti-pattern.

As described in Sect. 2, syntactic registries require words from the WSDL document describing a service to index it. This kind of registries operate 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 add meaningless or unrelated words to WSDL documents. *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 introduce meaningless or unrelated words.

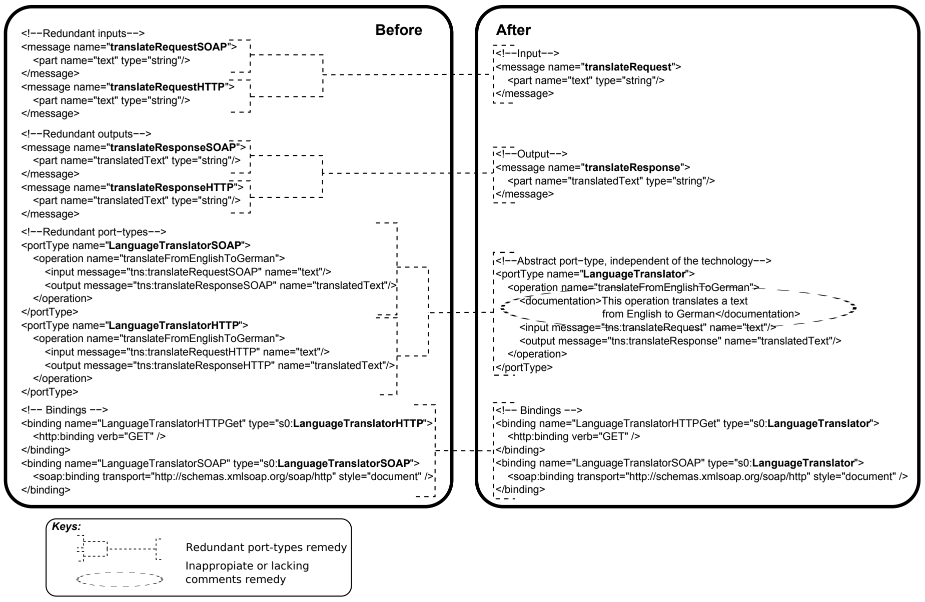
Another discoverability obstacle stems from the introduction of redundant words. As syntactic registries compute how significant a word is to a service, according to the occurrences of the word within the WSDL file [12,14,15], redundant words may be considered as an attempt to influence the ranking of 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 $word_W$ [12,14,15]. The anti-patterns which might affect the importance of words are

Table 1. WSDL Discoverability anti-patterns.

Anti-pattern	Concerns	Symptoms	Manifests	Problems	Remedy
Inappropriate or lacking comments	Documentation	Occurs when a WSDL document has no comments or comments are too complex to be understood.	Not immediately apparent	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 [8]	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 representatives 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 [19]	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 [20]	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.

Redundant port-types, *Empty message*, *Low cohesive operations in the same port-type*, *Redundant data models* and *Undercover fault information within standard messages*.

We have studied practical solutions to every anti-pattern described in Table 1. An anti-pattern remedy is a refactored design to improve WSDL documents discoverability. Refactoring is the process of rewriting any software code in order to improve it in any way. In this case, refactorizations aim at improving service discoverability. Specifically, for remedying anti-patterns that reduce the numbers 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 Fig. 2 depicts a Web Service description, whose


Before
After
Keys:
Redundant port-types remedy
Inappropiate or lacking comments remedy

```
<!--Redundant inputs-->
<message name="translateRequestSOAP">
   <part name="text" type="string"/>
</message>
<message name="translateRequestHTTP">
   <part name="text" type="string"/>
</message>

<!--Redundant outputs-->
<message name="translateResponseSOAP">
   <part name="translatedText" type="string"/>
</message>
<message name="translateResponseHTTP">
   <part name="translatedText" type="string"/>
</message>

<!--Redundant port-types-->
<portType name="LanguageTranslatorSOAP">
   <operation name="translateFromEnglishToGerman">
      <input message="tns:translateRequestSOAP" name="text"/>
      <output message="tns:translateResponseSOAP" name="translatedText"/>
   </operation>
</portType>
<portType name="LanguageTranslatorHTTP">
   <operation name="translateFromEnglishToGerman">
      <input message="tns:translateRequestHTTP" name="text"/>
      <output message="tns:translateResponseHTTP" name="translatedText"/>
   </operation>
</portType>

<!-- Bindings -->
<binding name="LanguageTranslatorHTTPGet" type="s0:LanguageTranslatorHTTP">
   <http:binding verb="GET" />
</binding>
<binding name="LanguageTranslatorSOAP" type="s0:LanguageTranslatorSOAP">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
</binding>
```

```
<!--Input-->
<message name="translateRequest">
   <part name="text" type="string"/>
</message>

<!--Output-->
<message name="translateResponse">
   <part name="translatedText" type="string"/>
</message>

<!--Abstract port-type, independent of the technology-->
<portType name="LanguageTranslator">
   <operation name="translateFromEnglishToGerman">
      <documentation>This operation translates a text
                     from English to German</documentation>
      <input message="tns:translateRequest" name="text"/>
      <output message="tns:translateResponse" name="translatedText"/>
   </operation>
</portType>

<!-- Bindings -->
<binding name="LanguageTranslatorHTTPGet" type="s0:LanguageTranslator">
   <http:binding verb="GET" />
</binding>
<binding name="LanguageTranslatorSOAP" type="s0:LanguageTranslator">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
</binding>
```

Fig. 2. Remedying a WSDL document: before and after eradicating anti-patterns.

offered operation lacks documentation, i.e., the WSDL document suffers from *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 shown within an ellipse in the right side of Fig. 2. As a result, now the operation description conveys relevant words, such as "translate", "text", "English" and "German", which might facilitate discovering this service through syntactical registries.

It is worth noting that the refactoring associated with the *Inappropriate or lacking comments* anti-pattern is in accordance 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 an unique and representative one. To clarify this, let us return to the service depicted on the left side of Fig. 2, which suffers from *Redundant port-types* anti-pattern as well. In this example, “LanguageTranslatorSOAP” and “Language-TranslatorHTTP” 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, Fig. 2, 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”). It is worth noting that it is possible to find two, or more, anti-patterns in the same WSDL document, as is the case of the example aforementioned. 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 representative words. Representative words describe the semantic of the elements that they refer to. Semantically, a representative 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 thing, 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. In the case of 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 [12,14,15], 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 [8].

The main goal of 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 implementation of the service, to completely eradicate the anti-pattern and guarantee that the WSDL document is a reliable description of the offered interface. Accordingly, it is worth noting that after applying the proposed remedies it is recommendable to verify whether the service functionality, which was working correctly before the refactorizations, keeps working as intended.

5 Data-Set Analysis

We have analyzed 391 publicly available WSDL documents [22]. Initially, we manually revised 130 files of the data-set and documented the catalog of WSDL discoverability anti-patterns of Sect. 4. To have an assessment of how common these bad practices are, we analyzed the whole corpus of WSDL files. Accordingly, each bar of

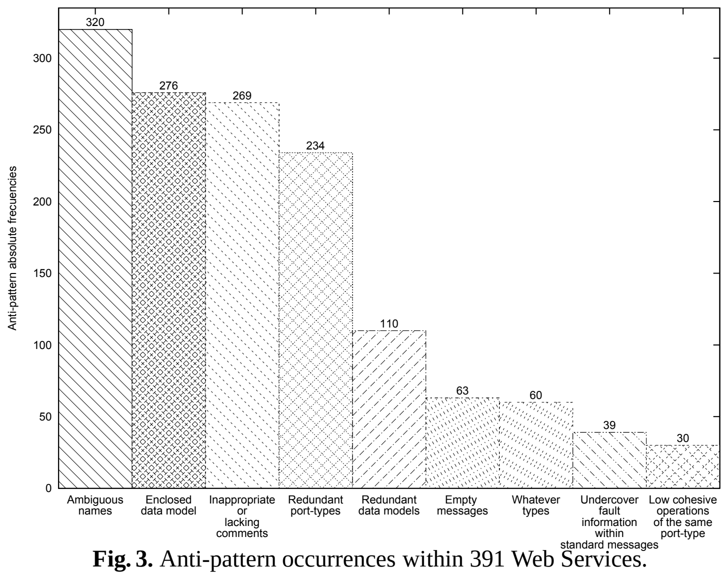


Fig. 3. Anti-pattern occurrences within 391 Web Services.

Fig. 3 depicts the number of WSDL files that suffer from an anti-pattern (not the number of anti-pattern occurrences). The reason to present the results in this way, is that some anti-patterns occur more than once in a WSDL document (*Ambiguous names*, *Empty messages*, *Redundant data models*, *Whatever types*, *Undercover fault information within standard messages*), but other anti-patterns 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.

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, though the intuitive importance of good naming and commenting practices, 82% and 69% of the documents suffer from the *Ambiguous names* and the *Inappropriate or lacking comments* anti-patterns, respectively. Additionally, the *Enclosed data model* anti-pattern, which not only resists discovery but also constrain data-model re-utilization, occurs in 70% of the documents. Similarly, notwithstanding port-types are meant to define the functionality of a service 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 suffer from the *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, an "Evident anti-pattern" provides visible signs in a WSDL document. 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" are *implicitly* present in the structure of a WSDL document, and these 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, "Present in a service implementation anti-patterns" have no footprint in a WSDL document, and they can be detected by invoking its implementation only. For instance, 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. In this case, *Undercover fault information within standard messages* anti-pattern can not 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.

6 Experimental Results

We have assessed the retrieval effectiveness of a syntactic registry using two versions of the data-set described in Sect. 5: one comprising the original WSDL documents, and another one comprising WSDL documents without anti-patterns. We built the improved version of the data-set by removing all the anti-patterns found. Our goal is to have an assessment of the discovery effectiveness improvements introduced by removing the anti-patterns of Sect. 4 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 [14,15]. Once the two data-sets were published, the same 30 queries, which are described in [15], were used to discover services through both registries. Finally, metrics were took to evaluate the impact of removing WSDL anti-patterns.

We decided to employ the measures summarized in Table 2, because they not only

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.	$\frac{RetRel_R}{R}$

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 the operations of a retrieved service fulfilled the expectations previously specified in each query or not.

We have calculated the aforementioned measures for each query and then averaged the results over the 30 queries. Figure 4 depicts the average results of each measure. In

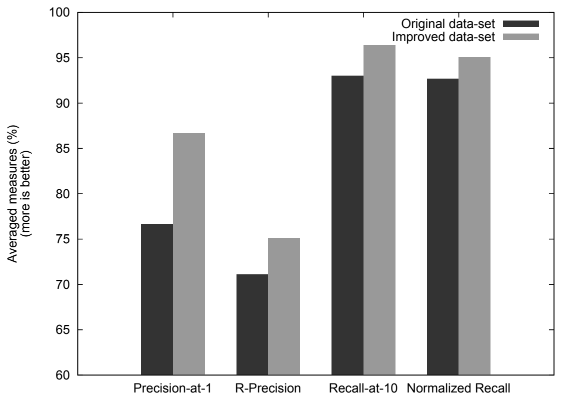


Fig. 4. Average measure results and how anti-patterns affect matching sensitivity.

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 4 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 representative names, the refactored WSDL documents have less terms, but they are more representative of the functionality of the services. The gain of 4% in *R*-precision results, has provided more evidence about 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 R^{th} position in the ranking, where *R* is the amount 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. Different experiments support that, because of users' tendencies to select higher ranked search results, even a small improvement in a rank has a great impact on discoverability [23]. 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% [23]. This further strengths the importance of removing discoverability anti-patterns from WSDL documents, in light of the significant Precision-at-1 improvements shown in the experiments.

7 Conclusions

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. 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 has presented 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. Opportunities for future evaluations include using other Web Service collections and other syntactic registries. Due to the effort required for manually improving large data-sets of WSDL documents, we are planning to develop heuristics for automatically detecting and remedying discoverability anti-patterns. Moreover, we are planning to research on measuring the implications of each individual anti-pattern on the performance of syntax-based search engines. To do this, we expect to exploit the aforementioned development on automatically eradicating the occurrences of each anti-pattern, separately from the rest of the other anti-pattern occurrences.

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.

This work will be extended in another direction to research on Web Service discoverability anti-patterns for 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 [24], thus these must be discovered to be exploited. Web Services are used to provide support for Grid infrastructures [25], 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 [26,27,28].

References

1. Bichler, M., Lin, K.J.: Service-Oriented Computing. Computer **39** (2006) 99–101
2. Erickson, J., Siau, K.: Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. Journal of Database Management **19** (2008) 42–54
3. McConnell, S.: Software Estimation: Demystifying the Black Art. Microsoft Corporation, Redmond, USA (2006)
4. Garofalakis, J., Panagis, Y., Sakkopoulos, E., Tsakalidis, A.: Contemporary Web Service Discovery Mechanisms. Journal of Web Engineering **5** (2006) 265–290
5. Sabou, M., Pan, J.: Towards semantically enhanced Web Service repositories. Web Semantics: Science, Services and Agents on the World Wide Web **5** (2007) 142–150
6. Hummel, O., Janjic, W., Atkinson, C.: Code conjurer: Pulling reusable software out of thin air. IEEE Software **25** (2008) 45–52
7. Paolucci, M., Sycara, K.: Autonomous semantic Web Services. IEEE Internet Computing **7** (2003) 34–41
8. Blake, M.B., Nowlan, M.F.: Taming Web Services from the wild. IEEE Internet Computing **12** (2008) 62–69

9. Song, H., Cheng, D., Messer, A., Kalasapur, S.: Web Service discovery using general-purpose search engines. In: IEEE International Conference on Web Services (ICWS). (2007) 265–271
10. Al-Masri, E., Mahmoud, Q.: Discovering Web Services in search engines. IEEE Internet Computing **12** (2008) 74–77
11. Nakamura, S., Konishi, S., Jatowt, A., Ohshima, H., Kondo, H., Tezuka, T., Oyama, S., Tanaka, K.: Trustworthiness analysis of web search results. In: Research and Advanced Technology for Digital Libraries (LNCS). (2007) 38–49
12. Dong, X., Halevy, A.Y., Madhavan, J., Nemes, E., Zhang, J.: Similarity search for Web Services. In: Proceedings of the 13th International Conference on VLDB, Toronto, Canada, Morgan Kaufmann (2004) 372–383
13. Stroulia, E., Wang, Y.: Structural and semantic matching for assessing Web Service similarity. International Journal of Cooperative Information Systems **14** (2005) 407–438
14. Crasso, M., Zunino, A., Campo, M.: Query by example for Web Services. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, New York, NY, USA, ACM (2008) 2376–2380
15. Crasso, M., Zunino, A., Campo, M.: Easy Web Service discovery: a Query-by-Example approach. Science of Computer Programming **71** (2008) 144–164
16. McCool, R.: Rethinking the Semantic Web, part II. IEEE Internet Computing **10** (2006) 96, 93–95
17. de Souza, C.R.B., Redmiles, D., Cheng, L.T., Millen, D., Patterson, J.: How a good software practice thwarts collaboration: the multiple roles of apis in software development. SIGSOFT Software Engineering Notes **29** (2004) 221–230
18. Henning, M.: API design matters. Communications of the ACM **52** (2009) 46–56
19. Pasley, J.: Avoid XML Schema wildcards for Web Service interfaces. IEEE Internet Computing **10** (2006) 72–79
20. Beaton, J., Jeong, S.Y., Xie, Y., Jack, J., Myers, B.A.: Usability challenges for enterprise Service-Oriented Architecture APIs. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). (2008) 193–196
21. Pendharkar, P.C., Rodger, J.A.: 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, New York, NY, USA, ACM Press (2002) 152–156
22. Heß, A., Johnston, E., Kushmerick, N.: ASSAM: A tool for semi-automatically annotating semantic Web Services. In: 3^{rd} ISWC. Volume 3298 of LNCS. (2004) 320–334
23. Agichtein, E., Brill, E., Dumais, S., Ragno, R.: 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, New York, NY, USA, ACM (2006) 3–10
24. Grimshaw, A., Morgan, M., Merrill, D., Kishimoto, H., Savva, A., Snelling, D., Smith, C., Berry, D.: An open Grid Services architecture primer. Computer **42** (2009) 27–34
25. Atkinson, M., DeRoure, D., Dunlop, A., Fox, G., Henderson, P., Hey, T., Paton, N., Newhouse, S., Parastatidis, S., Trefethen, A., Watson, P., Webber, J.: Web Service Grids: An Evolutionary Approach. Concurrency and Computation: Practice and Experience **17** (2005) 377–389
26. Mateos, C., Zunino, A., Campo, M.: JGRIM: An approach for easy gridification of applications. Future Generation Computer Systems **24** (2008) 99–118
27. Mateos, C., Zunino, A., Campo, M.: Grid-enabling applications with JGRIM. International Journal of Grid and High Performance Computing **1** (2009) 52–72
28. Mateos, C., Zunino, A., Campo, M.: A survey on approaches to gridification. Software: Practice and Experience **38** (2008) 523–556