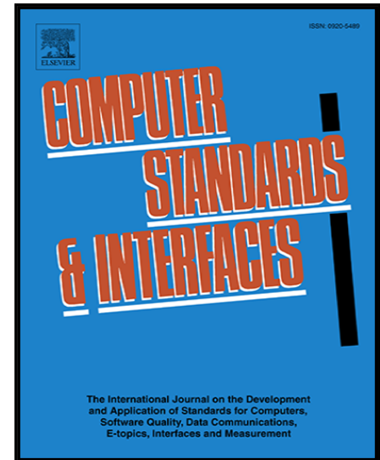
Accepted Manuscript



Spotting and removing WSDL anti-pattern root causes in code-first Web Services: A thorough evaluation of impact on service discoverability

Matías Hirsch, Ana Rodriguez, Juan Manuel Rodriguez, Cristian Mateos, Alejandro Zunino

PII: S0920-5489(17)30089-2
DOI: 10.1016/j.csi.2017.09.010
Reference: CSI 3243

To appear in: *Computer Standards & Interfaces*

Received date: 2 March 2017
Revised date: 29 September 2017
Accepted date: 29 September 2017

Please cite this article as: Matías Hirsch, Ana Rodriguez, Juan Manuel Rodriguez, Cristian Mateos, Alejandro Zunino, Spotting and removing WSDL anti-pattern root causes in code-first Web Services: A thorough evaluation of impact on service discoverability, *Computer Standards & Interfaces* (2017), doi: 10.1016/j.csi.2017.09.010

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

Highlights

- A code-first approach and tool support to develop discoverable Web Services in Java is described
- A thorough validation of the approach backed up by statistical significance tests is presented
- The GAnalyzer module early detects the presence of WSDL anti-patterns in service codes
- The GMapper module further reduces WSDL anti-patterns upon generating descriptions
- The assessment shows that the approach achieves better service discoverability levels compared to third-party approaches

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Spotting and removing WSDL anti-pattern root causes in code-first Web Services: A thorough evaluation of impact on service discoverability

Matías Hirsch[a,*], Ana Rodriguez[a], Juan Manuel Rodriguez[a], Cristian Mateos[a], Alejandro Zunino[a]

[a]*ISISTAN-CONICET, UNICEN. Campus Universitario, Tandil (B7001BBO), Argentina. Tel.: +54 (249) 4385682 ext. 35. Fax.: +54 (249) 4385681*

Abstract

To expose software as Web-accesible services, Web Service technologies demand developers to implement certain sofware artifacts, such as the service description using WSDL. Therefore, developers usually use automatic tools to perform this task, which take as input a code written in a programming language –e.g. Java– and generate the necessary artifacts for invoking it remotely. However, as a result of tool flaws and some bad coding practices, the description of the resulting Web Services might contain anti-patterns that difficult their discovery and use. In earlier work we proposed a tool-supported, code-first approach named Gapidt to develop Web Services in Java while early reducing the presence of anti-patterns in their descriptions through code refactorings. Bad coding practices, which potentially reduce the textual and structural information of generated WSDL documents, are automatically detected and informed to the developer by means of the GAnalyzer module so he/she can fix the service code. Moreover, developer provided information, such as service parameter names and operation comments, as well as re-utilization of data-type definitions, are exploited by the GMapper module upon generating WSDL documents. This paper focuses on a comprehensive experimental evaluation of the approach oriented at prospective users to assess expected discoverability gains and usage considerations taking into account various relevant service publishing technologies. In addition, we introduce a detailed comparison of Gapidt with a similar approach from the literature. The results show that Gapidt outperforms its competitor in terms of discoverability while improves Web Service description quality (better documentation and data-models). The Web Service discoverability levels of Gapidt outperforms that of third-party tools, either when using the GAnalyzer plus the GMapper, or only the GMapper.

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

1. Introduction

Service Oriented Computing (SOC) is a computing paradigm in which software running remotely offers functionality to other software. In SOC context, provided functionalities, which can be invoked through atomic operations, are called *services*. SOC defines three basic entities: *service provider*, *service consumer* and *service registry*. The service provider represents someone who offers services, while the service consumer represents someone that needs to invoke remote functionality. Providing and consuming services are not mutually exclusive activities as service providers can invoke third-party services to implement their own services, thus becoming prosumers [16]. Finally, the *service registry* acts as a link between providers and consumers. Basically, a service registry indexes services –offered by service providers– allowing service consumers to look for them.

With the exception of service registries, SOC entities are commonly implemented through Web Service technologies. These technologies enable software (service consumers) to remotely access third-party functionality [36], exposed by service providers, using widely-accepted protocols (HTTP, SOAP) and data

*Corresponding author.
Email address: matias.hirsch@isistan.unicen.edu.ar (Matías Hirsch)

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

interchange formats (XML, JSON), making this functionality interoperable. As a result, Web Services have been widely adopted in the industry. For instance, PayPal provides access to its payment services[1] through Web Services, and Amazon allows users to access its S3 Cloud Storage service through an SOAP-over-HTTPS API[2]. Moreover, in replacement of the UDDI specification, which was originally proposed to implement service registries but never experienced the expected adoption levels, the current approach to service discovery involves user-friendly online API repositories providing a keyword search-like interface of Web Services. Examples of such repositories are Mashape.com[3] and ProgrammableWeb.com[4]. Particularly, this latter hosts to date 17,173 Web Services and 7,645 mash-ups (composed services). Despite Web Service widespread adoption, different studies [6, 7, 45, 32, 43] have shown that in practice SOC developers tend to ignore good development practices when using Web Service technologies. Therefore, most Web Service interface definitions have low quality, which negatively impact on service consumers, service registries and service providers alike [35]. By *quality*, we mean how *understandable* a service is, i.e. how easy is for a service consumer to reason about a service functionality from its interface definition, or how *discoverable* a service is, i.e. how easy is for a service consumer to retrieve relevant services from a service registry upon issuing queries.

Since Web Services are intended to be invoked as black-box components, their only public artifact is their interface definitions, or simply interfaces. These interfaces are specified in an XML-based language called Web Service Description Language (WSDL)[5]. WSDL documents contain two parts, namely abstract part and concrete part. The abstract part describes the offered functionality, such as used datatypes and offered operations. The concrete part defines how to invoke these operations from a technical perspective. For instance, the abstract part of some service could define an operation getWeatherForecast with its input/output data (e.g. latitude/longitude as input, and high/low temperature, rain, snow, ice, and thunderstorm probability as output), while the concrete part defines that this operation can be called using SOAP plus HTTP or HTTPS and also specify the server address where the service is listening for incoming requests.

WSDL documents are intended to be both machine processable and human readable [40]. Therefore, the WSDL specification defines tags containing not only technical information, but also human readable information, such as identifiers and comments. However, it has been shown that in practice Web Service developers tend to disregard WSDL document textual information importance [45] and good APIs design principles [24, 17, 49, 39, 46]. Hence, service registries cannot effectively index Web Services, making them difficult to be discovered. Furthermore, automatic Web Service mash-up at the consumer's side can be negatively affected because there are composition techniques that rely on WSDL textual information [52]. Similarly to having low quality APIs [19], service consumers might face difficulties when trying to consume Web Services with poorly-written WSDL documents. Of course, service providers are affected since the Web Services they publish cannot be discovered or understood easily. Therefore, techniques and tools for assisting developers to implement high-quality Web Services are needed.

To address that, it is first necessary to consider how WSDL documents are written. The first option is to manually write WSDL documents. This methodology is called *contract-first* because WSDL documents are considered to be Web Service interface contracts: WSDL specification is guided by the intended service functional interface. Currently, there are a tools [44, 35] that allow service developers to verify whether their WSDL documents are affected by bad practices. The second option, which is commonplace in the industry, is to write the Web Service logic in a programming language first and then use an automatic tool for inferring the WSDL documents, so the WSDL-exposed functional interface "emerges" from the implemented Web Service code. This methodology is called *code-first*. These generation tools are widely used because developing Web Services using them is cheaper and faster. Moreover, these tools' main advantage is that developers do not need to have a deep knowledge of Web Service specification languages, namely WSDL and XSD (XML Schema Definition [6]). However, developers are not in direct contact with

[1] PayPal SOAP API: https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture/
[2] Amazon S3 SOAP API: http://docs.aws.amazon.com/AmazonS3/latest/API/APISoap.html
[3] https://market.mashape.com/explore
[4] https://www.programmableweb.com/category/all/apis
[5] http://www.w3.org/TR/wsdl20/
[6] http://www.w3.org/TR/xmlschema-2/

ACCEPTED MANUSCRIPT

WSDL documents, so they might not be inspected and corrected by developers prior to service publication and hence indexation in the service registry.

Particularly in the case of code-first service development, Ordiales Coscia et al. [32] have shown a high statistical correlation between Web Service source code written in Java and its WSDL document quality. In particular, such work has studied correlations between well-known Object-Oriented (OO) quality metrics [5] and an extensive service discoverability anti-pattern catalog [45] affecting WSDL discoverability. These anti-patterns are similar to other API design known issues [6, 21, 19, 24, 17, 49, 39], but they focus on the effect on discovering published WSDL documents in information retrieval (IR) based service registries [51].

Although Ordiales Coscia et al.'s work [32] studies suitable source code refactorings (mostly those from Fowler's catalog [15]) that reduce anti-patterns in generated WSDL documents, this work does not directly aim at addressing anti-pattern causes as we advocate in the present paper. Therefore, some anti-patterns might still affect WSDL document even after applying such refactorings. For example, the Weighted Methods per Class (WMC) metric is correlated with *LCO* anti-pattern (i.e. WSDL documents having some unrelated operations from a functional perspective). In this case, arbitrarily dividing the class into two classes would effectively reduce the WMC, but this can place unrelated methods within the same class, e.g. this refactor does not warranty *LCO* free WSDL documents. In addition, some of the proposed refactorings compromise other API quality aspects and only reduce the value of a particular OO metric. For instance, the Coupling Between Objects (CBO) metric is related with having data-types defined within the WSDL documents, which renders the WSDL document anti-pattern called *EDM* [32] (embedding data-type definitions in WSDL documents so that reuse from other WSDL documents is not feasible). Therefore, the authors propose to replace complex classes in the service API by strings or integers. As a result, only primitive data-types are required to use the Web Service, yet these data-types will hinder the original data-type –i.e. domain object– and probably the strings should be parsed in an ad-hoc manner by the consumer.

In contrast to Ordiales Coscia et al.'s approach [32], in earlier work [29] we have proposed Gapidt, an approach to detect the direct causes of WSDL anti-patterns in Java source code when building code-first services. This approach adapts WSDL document anti-pattern detection heuristics from [44] to Java source code, and associates simple source code refactorings to the heuristics. Moreover, in practice some anti-patterns are introduced by WSDL document generation tools rather than by the developer. Therefore, the approach implements a WSDL document generation tool that aims at reducing anti-patterns in generated WSDL documents. Concretely, in this paper we focus on reporting a thorough evaluation of the approach by measuring the impact of applying the studied refactorings on the resulting WSDL document discoverability around two IR-based service registries, while discussing expected gains/effort depending on the refactoring applied. The main goal is to provide prospective users empirical evidence and guidelines to exploit the approach in practice. Compared to the preliminary discoverability evaluation conducted in the work which originally proposed the approach [29], the contributions of this paper can be summarized as follows:

- We conduct a per-anti-pattern discoverability assessment, meaning that the individual effect on service discovery of applying each refactoring associated to the detected anti-patterns is quantified. In [29], only the effect on discoverability of applying or not *all* service code refactorings was assessed, which has a strong practical limitation.

- We base our contributed experiments on state-of-the-art Java to WSDL generation tools from popular Java-based Web Service development frameworks, namely Axis2, CXF2 and EasyWSDL. Moreover, since Gapidt proposes its own generation tool (GMapper), in addition to considering how Gapidt code refactorings help in early avoiding anti-patterns (see Section 4.4), we quantify the relationship between using GMapper and anti-pattern avoidance (see Section 4.3). This assesses whether using only the GMapper tool over commercial ones improves Web Service discoverability, as developers might not apply some Gapidt refactorings as they break service interface backward compatibility.

- Apart from WSQBE [45], we include in our analysis another IR-based registry, i.e. Lucene4WSDL [32].

- We include a detailed comparison of Gapidt with previous research [32] proposing a related early

ACCEPTED MANUSCRIPT

Anti-pattern name	Acronym	Symptoms	Manifestation
Enclosed data model	*EDM*	The type definitions are embedded in WSDL documents rather than placed in separate XSD ones.	Evident
Redundant port-types	*RPT*	Several port-types offer the same operations.	Evident
Redundant data models	*RDM*	The same domain object is mapped to different types in a WSDL document. This is mostly caused by defective WSDL generation tools [29].	Evident
Whatever types	*WT*	Some type definitions are based on the `any` XSD type.	Evident
Inappropriate or lacking comments	*ILC*	(1) a WSDL document has no comments (2) comments are inappropriate and non explanatory.	(1) Evident (2) Not immediately apparent
Low cohesive operations in the same port-type	*LCO*	Port-types have weak functional cohesion.	Not immediately apparent
Ambiguous names	*AN*	Ambiguous or meaningless names are used for denoting WSDL document elements.	Not immediately apparent
Undercover fault information within standard messages	*UFI*	Output messages are used to notify service errors: (1) type definitions and operation comments suggest anti-pattern occurrence. (2) it is necessary to analyze service implementation.	(1) Not immediately apparent (2) Present in service implementation

Table 1: WSDL anti-pattern catalog

service discoverability improvement approach. The experiments performed confirm that Gapidt outperforms the correlation-based approach (see Section 4.6).

- We backup all the experimental scenarios via statistical significance tests, particularly the Wilcoxon test, to ensure results validity.

The rest of the paper is organized as follows. Section 2 discusses relevant related works. Section 3 describes the Gapidt approach to WSDL anti-pattern detection causes for Web Services. The approach targets Java, a popular language in backend development. This Section also illustrates via examples the anti-pattern detection heuristics and associated refactorings, which are useful to allow the reader interpreting the experimental results reported later. Then, Section 4 explains the extensive experiments performed to both evaluate the approach and discuss the impact of the results in practice, which is the main goal of this work. The Section also reports the comparison of Gapidt against Ordiales et al.'s approach. Finally, Section 5 concludes the paper.

2. Background and related work

Several studies [37, 6, 7, 45, 32] have shown that in practice developers tend to disregard WSDL document quality. Particularly, [45] was the first work identifying and characterizing common bad practices that affect WSDL document discoverability. Some of these bad practices were partially studied previously, such as bad naming practices [7], bad commenting practices [14] or interface design flaws [37, 6]. The main contribution of [45] was categorizing these bad practices in an anti-pattern catalog, and both assessing their impact in terms of Web Service discoverability and proposing sound refactorings for each anti-pattern. In

ACCEPTED MANUSCRIPT

such work, the authors presents a detailed evaluation of how each bad practice impacts on the effectiveness of Web Service registries. The importance of this issue is beyond individual service discovery since mash-up techniques, which combine simple services to build added-value complex Web Services, also rely on service registries to search for suitable services [8, 52].

Table 1 presents these anti-patterns. In the table, the first three columns present the anti-patterns' given name, the acronym used through this paper for referencing each anti-pattern and a brief description. The fourth column, called "Manifestation", categorizes each anti-pattern according to how they can be detected. Particularly, there are three categories, namely "Evident", "Not immediately apparent", and "Present in service implementation". "Evident" anti-patterns can be detected on the WSDL document via simple algorithms: for instance detecting the *WT* anti-pattern consists on verifying whether a data-type is defined using the constructor `xsd:any`. "Not immediately apparent" anti-pattern requires cannot be detected by simple algorithms because requires interpreting the semantics of the analyzed elements, for instance, determining whether a name has the *AN* anti-pattern requires knowing if it actually depicts what is named. Finally, "Present in service implementation" means that there might not be evidence of such anti-pattern in the WSDL document. For instance, if the service suffers from the *WT* anti-pattern, error information might be conveyed within an output message, but there would be no evidence of that in the service WSDL document.

Discoverability anti-patterns do not only have a negative impact on service discovery, but also in Web Service understandability. In this context, a real-life case study [42] has provided evidence that these anti-patterns can be harmful even when discovery is not part of the client-side system development life-cycle. This case study is the modernization of a COBOL based system from a large Argentinean governmental agency system to a service-oriented system. During a first stage, developers generated the service using different automatic wrapping technologies. In particular, three automatic wrappers where used. The fist ones was for wrapping from COBOL to COM+, the second wrapped COM+ to C#, and the last exposed the C# wrapper as Web Services. Due to COBOL and wrapper technology limitations, the process resulted in WSDL documents with a large amount of anti-patterns. Despite being possible to call the functionality through Web Services, developers found very difficult to use those Web Services. As a result, the agency should develop services from scratch to make them easy-to-use. This is because the anti-patterns affect WSDL document understandability, impairing service consumers ability to easily understand service functionality and invoke service operations. This is expected because anti-patterns are analogous to well-known API issues in conventional software development [5, 21, 19, 9, 24, 17, 39, 49], such as naming, organization and comment problems, and WSDL documents can be seen as API definitions for Web Services.

In [18], the authors analyze the impact of well-known code-first tools for Java and .Net on the generated data-type definitions. The tools analyzed are JavaEE/Axis2, JavaEE/JAX-WS, JavaSE/JAX-WS, and .Net/WCF. According to the results, all Java tools lack support for at least one common data-type. For instance, according to the authors, JavaEE/Axis2 does not support char, ArrayList, abstract classes and Interfaces, among others, while JavaSE/JAX-WS does not support Map, Throwable, abstract classes, and Interfaces. Authors argue that such limitation have undesired side-effects, such as ambiguous data-type definitions or even forcing developers to manually define their WSDL documents. Briefly, [18] points out that even widely used WSDL generation tools have severe limitations regarding commonly used Java classes. Hence, in a second part of the same study [18], the authors outline an heuristic for detecting common ambiguous data-type specification in WSDL documents. Using their approach, they analyzed the WSDL documents generated with the different tools. All in all, they report that the percentage of vague data-types defined by each tool was: 43.40% (JavaEE/Axis2), 27.06% (JavaEE/JAX-WS), 16.07% (JavaSE/JAX-WS) and 29.53% (.Net/WCF). Furthermore, they analyzed 200 WSDL documents randomly selected from the 2500 WSDL documents in Al-Masri and Mahmoud data-set [3]. As a result of such analysis, authors reports that in 54% of the WSDL documents, data-type definitions were actually vague, so those Web Services cannot be effectively consumed by relying only in their WSDL documents.

Since API interface problems have a deep negative impact on software development [19], different researchers [11, 28, 21, 24, 49] have focused on techniques for detecting API issues and improving API quality. In this context, [11, 28, 35, 21] present approaches for assessing API naming quality. Particularly, [11, 28] present two semi-automatic approaches, in which names are manually mapped to a pre-defined defined set of concepts that can have relationships among them. Then, several properties are automatically tested to ensure conciseness and correctness. For instance, the same name should be associated to the same

ACCEPTED MANUSCRIPT

entity, otherwise ambiguity exists. Another desirable property is that a name of a specific concept should include the name of the more generic concept.

Moreover, the approach by Ouni et al. [35, 50] studies how to detect ambiguous names in service descriptions via search-based software engineering, and particularly a novel genetic algorithm. Based on 22 simple metrics that quantify different structural and semantic aspects of Web Service descriptions, and an input knowledge base of Web Services and their anti-patterns (ground truth), the algorithm produces rules in the form ***if*** *"combination of metrics with their threshold values"* ***then*** *"anti-pattern type"*, which can be employed to automatically detect anti-patterns in services outside the knowledge base. The fitness function is designed to maximize the number of detected anti-patterns w.r.t. the ones expected in the knowledge base. The work also illustrates the use of the same approach to detect anti-patterns related to service interface granularity (namely multiservice, nanoservice, chatty service) and whether a given service is a data service, i.e. a Web Service providing only accessor operations. Even when we also aim at fixing ambiguous names, and from a cohesion perspective the multiservice, nanoservice, chatty service anti-patterns relate to the *LCO* anti-pattern addressed in this paper, [35] targets contract-first Web Services.

In [34], the authors present an extension of [35] using Genetic Programming for detecting Web Service anti-patterns [35, 50]. This extension not only considers the five anti-patterns analyzed in [35], but also incorporates three more into the analysis, namely *Redundant PortTypes* [20], *CRUDy Interface* [12], and *Maybe It is Not RPC* [12]. As in their previous work, they use metrics for both WSDL documents and OO programming for predicting anti-pattern occurrences in WSDL documents. Since the WSDL documents were collected from the Internet, the authors have no access to the original Web Service source code. Hence, they generated a mock Java source code using JAX-WS[7], which the authors recognize this as a possible treat to validity. The evaluation data-set consists of 425 services divided into 10 categories. The experimental evaluation has shown that this approach performed better than the previous work. The analysis was performed over the whole data-set, as well as for different part of the data-set based on the categories of the data-set. In brief, the proposed approach obtained a median precision of 89% and median recall of 93%.

In a recent study, which was presented as a master thesis [38], the authors have analyzed the potential of machine learning techniques –particularly lineal regression, Gaussian process and multi-layer perceptron–for predicting the occurrence of anti-patterns in future versions of a Web Service. This work considers five anti-patterns, each of them defined by a name and a detection rule based on certain metrics over the WSDL document. Although it is not explicitly stated, such anti-patterns are the same five anti-patterns that Ouni et. al. aim at detecting in their work [35, 50]. Furthermore, the rules described in [38] are the same rules that Genetic Programming found in [35]. According to the results reported in [38], the lineal regression technique showed the best performance with precision and recall higher than 0.82 for both metrics. The main drawback of this study is that only eight Web Service where used for validating their approach, but this is to be expected as most WSDL document datasets only have one version of each WSDL document. However, these eight Web Services where from well-known enterprises, namely, Amazon, FedEx, Microsoft, and PayPal, and they have a rich release history with up to 44 releases of the same Web Service.

In contrast to [35, 11, 28], [21] proposes a completely automatic approach to name improvement, but the approach can only assess API method names. Firstly, the approach analyses method names using part-of-speech tagger based on WordNet [30] and an ad-hoc dictionary. The tagger basically maps the name of a method to its grammatical structure, e.g. the name `isFile(...)` can be mapped to <verb>-<noun>. Then, the approach automatically assesses methods properties, such as method return type, whether the method reads/writes instance variables, or if there is a loop within the method. Using the names, their tags and the method properties, the approach attempts to infer rules. For instance, a rule could be: when the method name matches `is*`, where * is a wild-card, the return type is boolean because in the 99% of the cases where the name started whit the word "is" the return type was a boolean. Therefore, any method whose name starts with "is", but does not return a boolean, is considered to be badly named. The authors argue that developers using that API would expect that naming patterns correspond with methods properties. Interestingly, the rules inferred vary when analyzing different software APIs.

[7]WSImport Tool used by the authors of [34]: `http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html`

ACCEPTED MANUSCRIPT

Regarding comments, [24, 49] present general techniques for assessing comment quality in source codes. In [24], the authors focus on Javadocs[8], which is the standard for commenting Java classes, and use different metrics to measure comment quality. Some of these metrics are only comment-related, such as Words Per Javadoc Comment Heuristic (WPJC) [48] or the older Flesch Reading Ease Level [25], which assess how difficult is a text to be understood. Other metrics consider the relationship between comments and source code elements, such as parameters names or exceptions thrown. Additionally, the authors evaluated the correlations between these metrics and the number of bugs found in different versions of ArgoUML and Eclipse.

Another known important issue in API design is the structural cohesion of its components [19, 9, 39]. In the pioneering work described in [5], class cohesion is measured using Lack of Cohesion on Methods (LCOM), or the number of method pairs without shared variables minus the number of method pairs with shared variables. Other cohesion metrics [9, 39] are based on representing classes as graphs and measuring properties of these graphs. These graphs have methods and variables as nodes, and the metrics can be seen as properties of these graphs. Although the metrics have been proved to be useful, certain type of methods, such as constructors and access methods (getters and setters) in service front-end classes, might negatively impact on their ability to successfully estimate the cohesion of a class [2]. In response, the work in [40] investigates how to quantify *semantic* cohesion of service APIs directly from WSDL documents. Concretely, the authors propose a metric to quantify document readability by extracting terms and determining their semantic distance in the WordNet tree. Experiments conducted with real users show a relation between WSDL document readability (as judged by humans) and the scores obtained by the proposed metric. The work also includes by-example guidelines to improve WSDL readability. As the authors indicate, however, the work is not suitable to code-first services.

Another approach to improve service descriptions from WSDL documents is [41], where the authors aim at rearranging operations within WSDL documents to increase cohesion. The proposed approach, called VizSOC, uses unsupervised machine learning techniques for identifying potential clusters of operations within a set of WSDL documents. Each operation is represented using a vector of term frequencies considering the terms which appear in the operation name, data-types, and data-type names. The authors have assessed four clustering techniques, namely K–Means, X-Means, PAM, and COBWEB. According to the reported results, COBWEB performed almost always better that the other clustering algorithms for all the considered metrics. Finally, this work proposes to visualize the clusters using Hierarchical Edge Bundles for allowing service developers to actually refactor the operations to increase the cohesion of their port-types and WSDL documents.

Regarding Web Service discoverability quantification, there is a tool [44] that aims at detecting the anti-patterns described in [45]. This is since [45] does not propose an automated way of spotting and/or correcting the various discoverability anti-patterns. The tool automatically analyzes WSDL documents searching for anti-patterns, and detects "Evident" anti-patterns by syntactically analyzing the elements of the WSDL documents. For instance, detecting *RPT* consists in comparing all the port-types and finding two port-types that have the same number of operations and with the same name[9]. However, detecting "Not immediately apparent anti-patterns" requires a semantic analysis of the documents, such as determining name structure. Although the approach in [44] is designed for performing a quality analysis directly over WSDL documents, it is not the best approach when developing Web Services using the *code-first* methodology because by definition developers have no direct control over the resulting WSDL documents. Then, a previous work [31] has found that there are significant correlations (*p-value* < 0.05) between some well-known OO metrics [5] and some discoverability anti-patterns. Therefore, instead of relying on contract-first WSDL inspection tools such as [44], code-first Web Service developers might increase the WSDL document discoverability by adjusting such code metrics. Since these are not perfect correlations, modifying the service code to indirectly produce a high impact on these metrics does not warranty the same

[8]Javadoc Tool: `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`

[9]WSDL *port-types* should not be confused with WSDL *port*s. A port-type is a named set of abstract operations and messages involved, while a port defines an individual endpoint by specifying a single address for a binding. Hence, a good practice is to define several ports that map a **single** port-type binding to several addresses (e.g. for increased availability), but duplicating a port-type definition (one per port defined) leads to the anti-pattern.

impact on WSDL discoverability.

Refactoring the source code based on OO metrics reduces anti-pattern symptoms in generated WSDL documents. However, sticking to the refactorings described in [31] might in turn produce other API design bad practices. For instance, a refactoring to diminish the WMC metric –which impacts on certain anti-patterns– is to divide the class into two classes, but blindly dividing a class might result in placing cohesive methods into two classes, violating well-known design principles [9, 2, 35]. Furthermore, representing all methods' parameters and return types using primitive types such as Strings, as the approach encourages, reduces the CBO metric. Yet this is the same as using the String type as a wild-card type, which is discouraged in both Web Services and OO programming [37, 4, 45]. Furthermore, since the anti-pattern detector presented in [44] does not consider String as a wild-card type, the anti-pattern detector would miss such String misuse.

Aiming at directly reducing anti-pattern causes in the source code might produce better WSDL documents in terms of understandability than the OO metric approach proposed in [31]. Even more, the former approach can help avoiding anti-patterns that are not considered by [31], such as text-related issues like the *ILC* anti-pattern. In this line, the study in [29] has shown that directly reducing anti-pattern causes improves resulting WSDL document discoverability. However, as pointed out in the Introduction, there are many differences between this study and the present paper. From a methodological point of view, we now study the effects of partially/completely early refactoring services codes or employing specialized WSDL generation tools on service discovery. The study in [29] is solely focused on achieving maximum discovery in the sense that performed experiments assessed the impact of completely refactoring codes and maximizing WSDL generation tool usage, without paying attention to other important issues such as assessing the effects of individual refactorings, discussing refactoring effort, and comparing the effects of refactorings and/or using specialized tools. On the other hand, in the present paper, we aim at assessing these aspects by evaluating Gapidt while including more service registries (i.e. Lucene4WSDL [32]) and other state-of-the-art Java-based WSDL generation tools (e.g. EasyWSDL[10]) to produce comprehensive practical results.

3. Spotting and removing WSDL anti-pattern root causes: The Gapidt approach

This section, outlines the Gapidt approach for Java code-first Web Service development already proposed in [29], which aims at reducing the impact of the WSDL discoverability anti-patterns first identified in [45] on the generated WSDL documents. The approach consists of two phases. The first phase is identifying and refactoring bad practices associated with the anti-patterns from the Java source code that the developer intends to expose as a service. For this step, a tool called GAnalyzer assists the developer in detecting such bad practices. The detection techniques that this tool implements are based on proven techniques for detecting such anti-patterns in WSDL documents, originally implemented in a discoverability anti-pattern detection tool for contract-first services [44]. The second phase provides a WSDL document generation tool, called GMapper, which works in a similar fashion as third-party tools, such as that of Axis2[11] or EasyWSDL. However, GMapper has been designed for transferring as much textual information to the WSDL document as possible, and avoiding other source of anti-patterns, such as replicating the port-types for every supported communication protocol. For instance, Axis2 Java2WSDL does not map Javadoc comments to WSDL operation documentation resulting in the *ILC* anti-pattern, while GMapper does it. Although mapping the comments into the WSDL document does not guarantee that the WSDL document is not affected by the *ILC* anti-pattern because the Java code might lack comments or they might be inappropriate, the anti-pattern would not result from using GMapper.

Figure 1 outlines the options that a developer has, considering the Gapidt approach, to develop code-first services. The first option –illustrated by the pipeline at the bottom of the Figure– is to fully exploit the approach by firstly analyzing the code using GAnalyzer, manually refactoring it, and then generating the WSDL documents using GMapper. A second option is not to analyze the code, but using GMapper for

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

[11]Axis2 Java2WSDL: `https://axis.apache.org/axis2/java/core/tools/maven-plugins/maven-java2wsdl-plugin.html`

ACCEPTED MANUSCRIPT

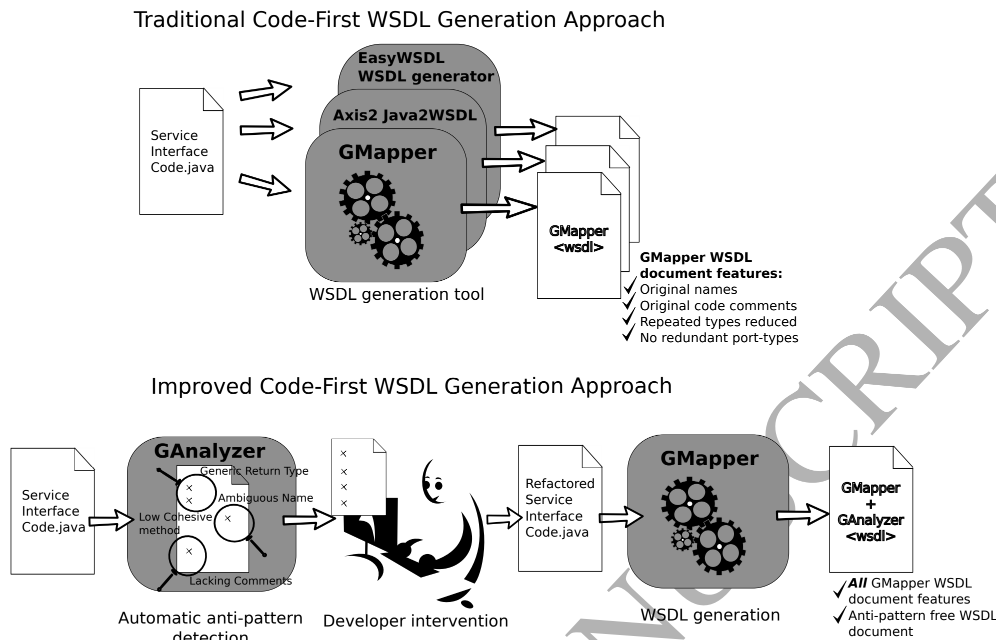


Figure 1: Gapidt approach overview

generating the WSDL documents. Third, developers might opt for ignoring both GAnalyzer and GMapper and write code-first Web Services using only existing generation tools. Actually, the second and third options represent the traditional approach to Web Service development, where service codes are fed to a WSDL generation tools without introducing early discoverability improvements. Notice that GAnalyzer is not capable of automatically refactoring the code, and the code refactoring task is responsibility of the developer. In order to facilitate the integration of the Gapidt approach into the service development and publication process, both GAnalyzer and GMapper have been materialized as an Eclipse plug-in, called Gapidt. This software is available upon request.

Gapidt Eclipse plug-in has not only been designed for enforcing the Gapidt approach, but also for assisting Web Service code-first development in general. For this reason, using GAnalyzer is optional. Even more, Gapidt Eclipse plug-in does not only support GMapper to generate WSDL documents associated to services, but also two of the most common third-party tools, namely Axis2 Java2WSDL and EasyWSDL WSDL generator.

The rest of the section describes in detail the components for anti-pattern cause detection and WSDL document generation of the tool. Particularly, Section 3.1 describes GAnalyzer, while Section 3.2 presents GMapper. It is worth noting that the contents of both subsections are mostly derived from [29], with the goal of making the paper self-contained and provide the reader the necessary ground to correctly interpret the experimental results shown later in the paper.

3.1. GAnalyzer

GAnalyzer detects certain bad practices that might result in anti-patterns when generating WSDL documents. Despite being capable of analyzing any Java class, GAnalyzer is intented to be used on the service front-end class that would be exposed as a Web Service. In particular, GAnalyzer is capable of detecting practices associated with five discoverability anti-patterns, namely *ILC* , *AN* , *LCO* , *WT* and *UFI* anti-patterns. This is because not all anti-patterns are directly related with coding practices. For instance, *EDM* anti-pattern occurs when the automatic mapper places type definitions within the generated WSDL document. Although a previous study [32] has shown that defining complex data-types is associated with the *EDM* anti-pattern, the proposed solution in that work is to use primitive data-types for defining the Web Service API. This solution is not suitable in real projects because it conceals the data type objects of APIs.

ACCEPTED MANUSCRIPT

```
//Output Class
public class Book {
 private String title;
 private String ISBN;
 private String author;
 private int pages;
 ....
 //Getters and Setters
 ....
}

//Service front-end class
public class BookStoreCatalog {
 /** Service for costumers.*/
 public String getBookTitle(String ISBN){...}
 /** Returns a book given its ISBN.*/
 public Book getBook(String ISBN) {...}
 /**Retrieves a CD information using the store SKU.*/
 public CD getCD(int sku) {...}
}
```

```
//Output Class
public class CD {
 private boolean nonExistingCDError;
 private String artist;
 private String album;
 private int numTracks;
 private Track[] tracks;
 ....
 //Getters and Setters
 ....
}
```

Figure 2: Example Web Service: Java code

To clarify the proposed code analysis techniques of this component, we will use the same example through this section. This example consists of a fictitious Web Service that offers access to the catalog of an on-line CD/Book store. The code presented in the Figure 2 is part of the Java source code for the example Web Service.

For the sake of clarity, this section has been further divided into a sub section for each anti-pattern that can be detected.

3.1.1. ILC *anti-pattern causes' detector*

To detect the causes of *ILC* the tool firstly verifies whether all methods in the service front-end class are associated to comments in Javadoc form, which is the standard for documenting methods in Java. Then, if some method lacks comments, the problem is informed to the developer. Secondly, for all the commented methods, the tool analyzes the existing comments and compares them with the method name and the names of method parameters. Specifically, the tool generates two tree structures for the method name and method parameter names respectively, and two hypernym trees for verbs and for nouns in the method comment. Finally, the tool determines the similarity between the comment verb trees and method verb trees, and the similarity between comment noun trees and method noun trees. In this way, according to previous experiments, a low value of similitude is enough to ensure that comments and operation names are related [44] because the two trees usually have a very dissimilar number of nodes (the similitude tends rapidly to 0 when the sentence topics are unrelated). Therefore, if the similarity is less than 10% the tool reports the associated issue.

Let us explain the algorithm in more detail. The trees are constructed as follows:

1. The tool tags words in the text using a probabilistic context free grammar (PCFG) parser [10] and then extracts two sets, namely nouns and verbs.
2. For each set, the tool obtains all the hypernym words using WordNet [30], and organizes them from the most generic to the most specific word.
3. For each hypernym set, the tool combines the elements into a tree.
   (a) The tree starts with one ROOT node that is not associated to any concept.
   (b) For each hypernym list; for each element in the list
      i. if the element is in the tree, do nothing

ACCEPTED MANUSCRIPT

ii. else, add the element as child of its hypernym; if it has no hypernym, add it as child of the ROOT node.

The method `getBookTitle(String ISBN)` will be used to illustrate how the tree is constructed. The first step consists in separating the words in the method name using upper case characters as word beginning markers, namely "get", "book", "title", and adding "ISBN" (parameter name). The second step consists in retrieving from WordNet the hypernym lists for these words. As "ISBN" is not defined in WordNet, this word is therefore ignored. For the sake of brevity, we have only considered two hypernym lists for "book" and one for "title". Each list goes from the most generic concept to the most specific one:

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

Figure 3 shows the hypernym tree obtained using the hypernym lists described above as input, i.e. it is a partial tree for the words "book", "title" and "get". Notice that common concepts are subsumed in one node, e.g. there is only one node for the "entity" concept.

A particular word can have different sets of hypernyms because a word can have several meanings, while some words might have no hypernyms because they are not defined in WordNet. All in all, the similarity between two trees t_1 and t_2 is calculated as follow:

$$sim(t_1,t_2) = \frac{depth(sharedSubTree(t_1,t_2))}{max(depth(t_1),depth(t_2))} \quad (1)$$

, or the quotient between the depth of the maximum common subtree of t_1 and t_2 over the maximum depth of the two input trees.

3.1.2. AN *anti-pattern causes' detector*

To detect *AN* anti-pattern causes the tool applies a three-step heuristic. The first step verifies whether the names of classes, attributes and method arguments have an appropriate length, which should fall between 3 and 30 characters [44]. In a second step, the names are contrasted against a known unrepresentative name patterns list*: param, arg, var, obj, object, foo, input, output, in#, out# and str#*;, where # is a number. The third step consists in analyzing the names using the PCFG parser. Parameter names are expected to be "nouns" or "noun phrases", while operations names should be in the form "verb+nouns" because this has been proved to be a best practice in naming [11, 21]. Aiming at improving PCFG accuracy for operation names, the tool adds "it must" at the beginning of them. Note that the example is not affected by these anti-pattern causes. First, the length of all the names is between 3 and 30 characters. Secondly, no word within the names is in the unrepresentative name list. Regarding name structure, all the operations names have the form form "verb (get) + noun (book title, book author, CD)", while all the parameter names are nouns, such as ISBN, or SKU (Stock-Keeping Unit).

3.1.3. LCO *anti-pattern causes' detector*

Thirdly, to detect *LCO* anti-pattern causes, the tool creates an undirected graph that represents the relation among the methods present in the service front-end class and the data-types. An example of a cohesion graph is depicted in Figure 4. This graph has three kinds of nodes:

ACCEPTED MANUSCRIPT

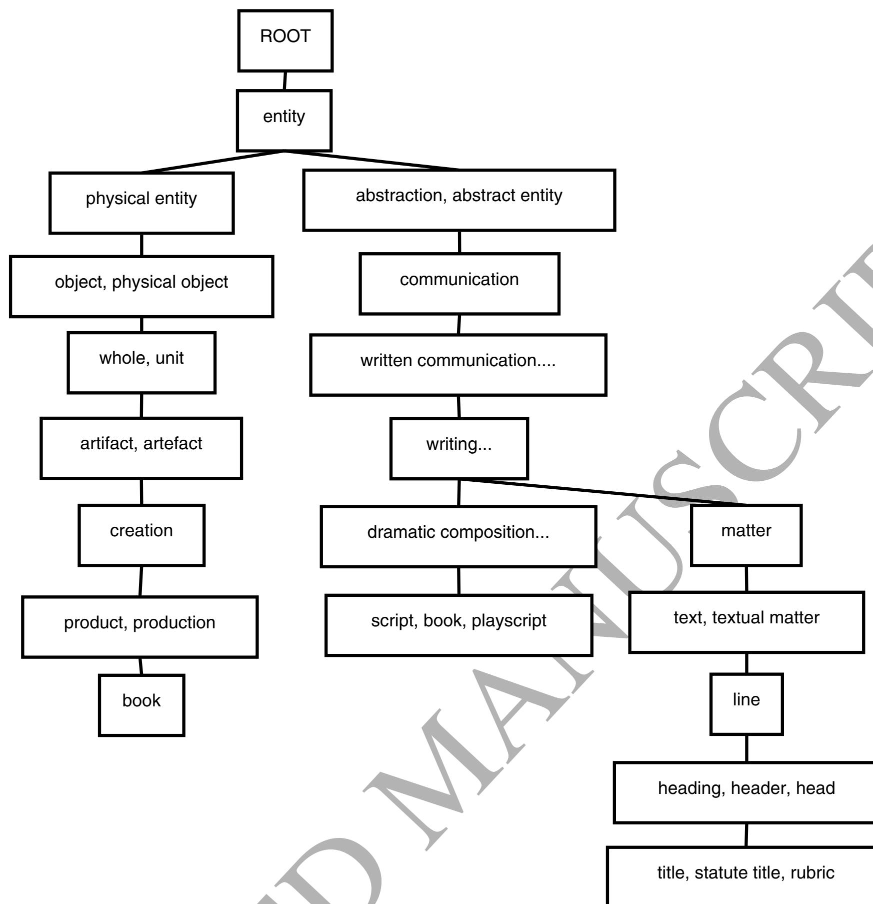


Figure 3: Example hypernym tree

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

Two nodes might be connected by one of the following edges:

- one node has an attribute of the type of another node: it represents a type having an attribute of another type.
- parameter: one method receives as input or returns a particular type.
- parametrized with: a type is parametrized with another type, e.g. List<String>. It also represents relationships through ignored types, such as arrays or maps.
- noun in the name: this is the relation between a method and all the nouns included in its name.

Using the generated graph, the tool determines whether this graph is fully connected. Otherwise, the tool selects the largest connected subgraph and reports methods not included in this subgraph as non-cohesive methods, e.g. the method getCD in the example.

ACCEPTED MANUSCRIPT

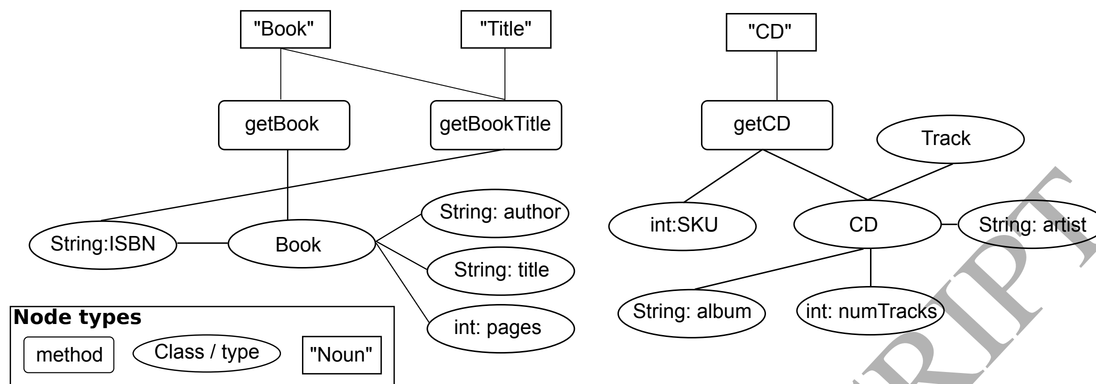


Figure 4: Example cohesion graph

3.1.4. WT *anti-pattern causes' detector*

To detect *WT* anti-pattern causes, the tool uses a known list of too generic classes [4]. When a method parameter or return type is one of these classes, the tool reports the issue. The classes in this list are Object, Vector, List, Map, Collection, Enumeration, Vector<Object>, List<Object>, Map<Object, Object>, Collection<Object> and Enumeration<Object>. The example is not affected by this anti-pattern causes because lists are correctly parametrized and no method receives as input or returns as output an instance of Object. A variation of the `getBook` method affected by this anti-pattern cause might be one which returns an instance of Object.

In practice, this problem commonly manifests when programmers do not properly use Java generics. Modern IDEs such as Eclipse warn the programmer about this situation. Correctly parametrizing classes and data structures avoid the mapping of generic types to the xsd:any XSD construct.

3.1.5. UFI *anti-pattern causes' detector*

Finally, to detect *UFI* anti-pattern causes, the tool has to analyze the method output structure. In this way, the tool first verifies whether the method has exceptions defined or not. If the method has no exception defined, the tool analyzes the output class field names looking any of the following keywords: *"ping"*, *"error"*, *"errors"*, *"fault"*, *"faults"*, *"fail"*, *"fails"*, *"exception"*, *"exceptions"*, *"overflow"*, *"mistake"*, *"misplay"*. Using one of these names suggest that the output might convey error information, which should be returned using an exception. In the example, the method `getCD` is affected by this anti-pattern causes because the CD class has a boolean that indicates whether the CD was found. A suitable refactoring would be to remove the boolean from the CD class and throw an `Exception`, e.g. developers could define a new exception named `NoSuchCDException` and redefine the method to throw it when a CD is not found in the database.

3.2. GMapper

After implementing the Web Service logic, GMapper can be used for generating the WSDL document. Notice that other similar generation tools, such those provided by Axis2 or EasyWSDL, can be used in this step, but GMapper is designed for reducing the impact of the anti-patterns in the generated WSDL documents. GMapper expects as input a Java class –source code– that implements the Web Service operations as public methods, i.e., the input class is a front-end class for the Web Service. Unlike Axis2 Java2WSDL, GMapper uses the Java source code because when compiled to Java bytecode, a Java class losses information, such as comments or some identifier names. This happens because although being useful for the developer, such information is not useful for the Java Virtual Machine that runs the bytecode. Using this class, GMapper builds a WSDL document by mapping each public method of a service front-end class to an operation within a port-type.

In order to avoid *ILC* anti-pattern, the Javadoc comment associated to a method is mapped into an operation comment, while the class Javadoc is mapped to a comment for the Web Service. For each operation, GMapper defines an input, an output message and as many fault messages as exceptions that

ACCEPTED MANUSCRIPT

```
<definitions...>
  <import namespace="..." location="..." />
  <message name="getBookTitleRequest">
    <part name="parameters" element="xsd1:getBookTitleRequest" />
  </message>
  <message name="getBookTitleResponse">
    <part name="parameters" element="xsd1:getBookTitleResponse" />
  </message>
  ...
  <portType name="BookStoreCatalog">
    <operation name="getBookTitle">
      <documentation>Service for costumers.</documentation>
      <input message="tns:getBookTitleRequest" />
      <output message="tns:getBookTitleResponse" />
    </operation>
    ...
    <operation name="getCD">
      <documentation>Retrieves a CD information
            using the store SKU.</documentation>
      <input message="tns:getCDRequest" />
      <output message="tns:getCDResponse" />
    </operation>
  </portType>
</definitions>
```

Figure 5: WSDL generated for the example Web Service

might be thrown by executing the method. For the case of the input, the name is preserved within the operation. On the other hand, the outputs in Java are not named, while they must have a name within a WSDL document, otherwise the *AN* anti-pattern arises.

Finally, the classes of the inputs and output of the methods are mapped to type definitions in XSD. GMapper assumes that method inputs and output classes are Plain Old Java Objects (POJOs) with properties only accessible through getters/setters. In this sense, GMapper can only map POJO attributes to data-types, and all defined behavior within a return class would be loss during the mapping. GMapper recursively follows class dependency graphs until it finds a type that can be directly mapped to an XSD type, such as int, double, float, String, or array. Regarding fault messages, the underlying mapping mechanism is similar, but the associated classes are expected to extend from java.lang.Exception.

Figure 5 presents part of the WSDL document generated by the tool for the Java code presented in Figure 2, while Figure 6 depicts the XSD schema imported by the WSDL document. Notice that this component helps avoiding the anti-patterns presented in Table 1, like *EDM* , while reducing the impact of other anti-patterns (e.g. *AN* or *UFI*) requires following the refactoring guidelines discussed in the previous section prior to WSDL generation. On the other hand, even when following the enforced good practices –e.g. commenting the source code– if developers do not use GMapper there is no warranty that the generated WSDL document will not be affected by the anti-pattern related to these practices. For example, when using WSDL document generation tools which disregard the comments in the source code, the generated WSDL documents would be still affected by the *ILC* anti-pattern. Finally, as stated above, GMapper aims at transferring as much textual information from the Java code to the WSDL document as possible, such as comments and names, but WSDL defines textual information that is not present in Java codes, such as returning type identifier. Therefore, not all anti-patterns can be avoided even when Gapidt practices and tools are applied and used.

4. Evaluation

This section describes the experimental setup, metrics utilized and results obtained from the empirical tests conducted, which aim at thoroughly evaluating GAnalyzer and GMapper. Basically, we study the impact, in terms of degree of discoverability, of using the Gapidt approach/tool or not to generate and refactor WSDL documents. We evaluated the impact of both refactoring service codes based on anti-pattern causes detection and usage of GMapper, for a wide range of discoverability tests. Furthermore,

ACCEPTED MANUSCRIPT

```
<xsd:schema ... xmlns:xsd="...">
 <xsd:element name="getBookTitleRequest">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="ISBN" type="xsd:string" minOccurs="0" />
  </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
...
 <xsd:complexType name="CD">
  <xsd:sequence>
   <xsd:element name="nonExistingCDError" type="xsd:boolean" />
   <xsd:element name="album" type="xsd:string" />
   <xsd:element name="tracks" type="tns:ArrayOfTrack" />
   <xsd:element name="artist" type="xsd:string" />
   <xsd:element name="numTracks" type="xsd:integer" />
  </xsd:sequence>
 </xsd:complexType>
 ...
</xsd:schema>
```

Figure 6: XSD of the example Web Service

some of these tests do not include code refactorings and aim at assessing the discoverability gains of GMapper over talternative popular WSDL document generation tools. Broadly, our discoverability tests consist in performing queries to an IR-based Web Service Registry populated with WSDL documents from two different data-sets and measuring discoverability through IR metrics.

The next sections are organized as follows: Section 4.1 provides details of the experimental setup, i.e., registries used, data-sets employed and discovery tests. Section 4.2 describe the IR metrics used to evaluate discoverability. Section 4.3 and Section 4.4 present the results and a detailed analysis of discoverability tests.

4.1. Experimental setup

To perform the discoverability tests, two IR-based Web Service Registries –from now on registries– named WSQBE (Web Service Query by Example) [45] and Lucene4WSDL [32] were used. WSQBE was designed to tackle Web Service indexing issues, such as joining words and WSDL document tags, dealing with the presence of technical words and low amount of relevant terms. WSQBE takes advantage of classification techniques to reduce the search space. Moreover, Lucene4WSDL is an adaptation of the well-known Lucene[12] library and indexes WSDL documents by extracting relevant terms. It assumes that element names, such as operation names or parameter names, might be composite names, so it splits them assuming *camelCase* notation and *underscore_term_separation* in the same way WSQBE does.

The registries, in each discoverability test, were populated with two data-sets. One data-set, the Relevant Data-Set, contains WSDL documents derived from 81 Java service implementations (particularly service front-end classes) including 637 service operations. These WSDL documents are relevant to the queries performed to the registry. The other data-set, the Noise Data-Set, contains WSDL documents that are not relevant to queries. These are included as noise, with the aim of simulating the content of a typical Web Service registry where documents from different domains, or from the same domain but exposing different contracts, coexist.

The Relevant Data-Set contains two WSDL documents for each Web Service. One of the documents belongs to the *Gapidt data-set* and was generated from a) the original service source code using GMapper (experiments of Section 4.3) or alternatively b) the refactored service source code with GAnalyzer plus GMapper (experiments of Section 4.4). The other document belongs the *Third-party data-set* and was generated from the original service source code using a third-party WSDL generation tool. Particularly, we used the WSDL generation tools of Axis2 Java2WSDL and EasyWSDL WSDL generator which are very

[12] Apache Lucene: `http://lucene.apache.org/core/`

Issue	# of affected Web Services	# of affected elements	Type of affected elements
AN	44	303	method/ parameter names
LCO	20	44	classes
ILC	78	505	methods
WT	7	23	parameter/return types

Table 2: Experimental Web Services and issues: Outline

popular in the industry for building Java-based Web Services [32]. For the sake of simplicity, from now own, we will use "Axis2" and "EasyWSDL" not to refer to the service frameworks themselves, but to their built-in WSDL generation tools, i.e., Axis2 Java2WSDL and EasyWSDL WSDL generator.

In fact, in our experiments we are not interested in using different Web Service frameworks (Axis2, EasyWSDL, CXF2[13] and so on) but rather different Java to WSDL conversion tools which lead to **varied** textual content in generated WSDL documents, which in turn might unevenly affect index-based service registries performance. By "varied" we mean WSDL documents having different useful words or the same useful words with different number of occurrences after being processed by a service registry. Upon indexing a new WSDL document, index-based registries –e.g. WSQBE and Lucene4WSDL– apply a battery of common text processing tasks to clean the document. Among these tasks is stopword removal. A stopword is a non-relevant word with a low level of usefulness within a given context of usage. For instance, WSQBE uses a list of about 600 English stopwords (e.g. prepositions and articles) plus a small list of stopwords related to the Web services domain, such as "request", "response", "soap", "post", "port", "type", and so on. With this in mind, we have not considered other popular Web Service frameworks in general and CXF2 in particular, since the WSDL generation tools used to generate service descriptions from Java source codes lead to the same textual content (bag of words) from the point of view of service registries.

With respect to the keyword-based queries to the registries, they were created using service operation names of WSDL documents in the Relevant Data-Set. Therefore, each query has two relevant WSDL documents associated, one from the Gapidt data-set and the other from the Third-party data-set. In other words, each WSDL document in the Relevant Data-Set is relevant to the same number of queries as operations it exposes.

There are two instances of third-party data-sets; one contains WSDL documents generated with Axis2 and another generated with EasyWSDL. Furthermore, there are seven instances of Gapidt data-sets, one contains WSDL documents derived from original source codes, four derived from applying individual refactorings to the service source code, and two derived from applying multiple refactorings to the service source code (one where all refactorings were applied and another where all refactorings but *ILC* refactoring were applied). The last Gapidt data-set was considered because comments in WSDL documents heavily increases the amount of text that the registry has to index, which might lead to quite different discovery performance. Table 2 shows the number of Web Services where anti-patterns were detected by the Gapidt approach, and the number and type of affected elements. These elements vary according to the anti-pattern: in the case of *ILC* the elements are methods, while in the case of *AN* the elements are methods and parameters names. Since not all anti-patterns were detected in all Web Services, the Gapidt data-sets include some non-refactored Web Services.

All discovery tests were performed for eighteen different Noise Data-Sets that were built by combining WSDL documents from two known sources: [20] integrated by 393 WSDL documents and [3] integrated by 1,664 WSDL documents. Noise Data-Sets sizes range from 393 to 2,057 WSDL documents. This is, Noise 1 data-set contains WSDL documents only from [20], Noise 2 data-set was built by combining Noise 1 data-set plus a hundred of WSDL documents from [3], Noise 3 data-set was built by combining

[13]`http://cxf.apache.org`

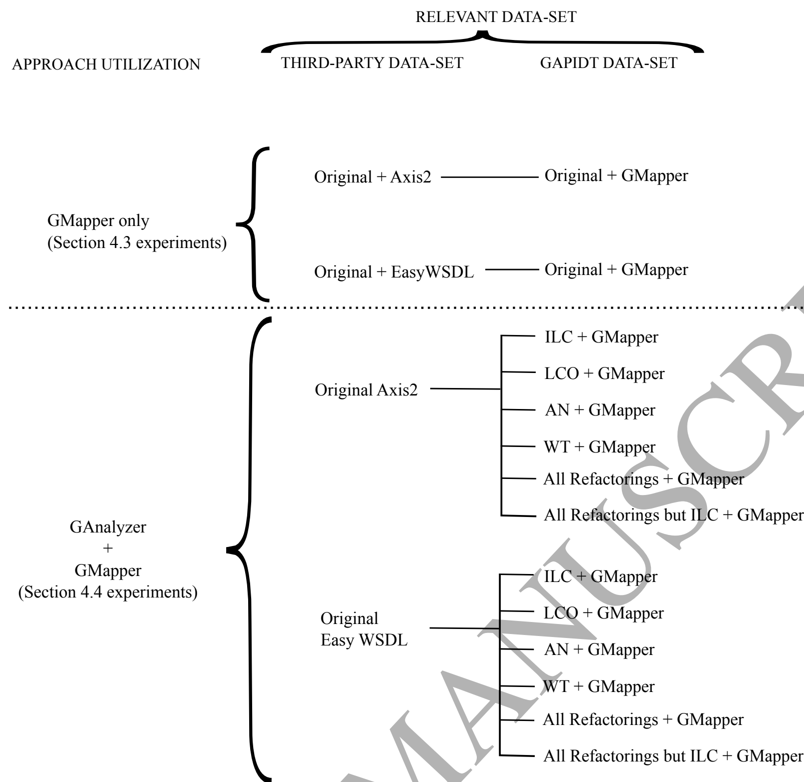


Figure 7: Outline of discoverability tests

Noise 2 data-set plus two hundred WSDL documents from [3], and so on. Lastly, Noise 18 data-set contains all documents from [20] and [3].

Registry	Relevant Data-Set		Noise Data-Set
	Third-party Data-Set	Gapidt Data-Set	
WSQBE, Lucene4WSDL	Original code Axis2, Original code EasyWSDL	Original code GMapper, *AN* GMapper, *LCO* GMapper, *ILC* GMapper, *WT* GMapper, All refactorings GMapper, All refactorings but *ILC* GMapper	Noise 1 (393 WSDLs), Noise 2 (493 WSDLs), Noise 3 (593 WSDLs), ..., Noise 17 (1,993 WSDLs), Noise 18 (2,057 WSDLs)

Table 3: Variables and values of discoverability tests

Table 3 summarizes the values (rows) of each experimental variable (columns) present in a discoverability test, i.e., registry, the Relevant Data-Set –integrated by a Gapidt data-set and a Third-party data-set–, and the Noise Data-Set. Figure 7 outlines the value combinations that were exercised as a discoverability test. The tests were grouped into those that aim at evaluating the discoverability of GMapper data-set against third-party data-sets, named GMapper tests, and those that aim at evaluate the synergy of GAnalyzer and GMapper data-sets against third-party data-sets, named GAnalyzer+GMapper tests. It is worth mentioning that each discoverability test shown in Figure 7 was, in fact, performed for all Noise Data-Sets and registries. The results and analysis will be presented by following the test order outlined in Figure 7 (from top to bottom and left to right). Then, Section 4.3 presents the discoverability evaluation of GMapper, while Section 4.4 the evaluation of GAnalyzer plus GMapper.

4.2. Discoverability metrics

We measured discoverability through nDCG and Recall metrics. The nDCG [22] metric strongly penalizes rankings with no relevant results in the first places of the results window. Formally:

$$nDCG = \frac{DCG}{IDCG},$$
$$DCG = \sum_{i=1}^{p} \frac{2^{rel_i}-1}{log_2(i+1)},$$

where rel_i is the relevance value of the document in the ith position of the ranking, and $IDGC$ is the DCG for the best possible ranking, i.e. the documents are ranked in descending order according to their relevance value. In our tests, a document is either relevant or not to a query, meaning that rel_i can be either 1 or 0. The $IDCG$ is the DCG for the ranking that places that document in the first place:

$$IDCG = \frac{2^1-1}{log_2(1+1)} + \frac{2^0-1}{log_2(2+1)} + .. + \frac{2^0-1}{log_2(n+1)} = 1$$

from which the next equation is deduced:

$$nDCG = \frac{DCG}{1} = DCG = \sum_{i=1}^{p} \frac{2^{rel_i}-1}{log_2(i+1)}$$

and there is only one non-zero term in the addition. This term corresponds to the position where the relevant document is ranked. As a result, in our discoverability tests $nDCG$ can be expressed as follows:

$$nDCG = \begin{cases} 0 & if\ relevant\ not\ found \\ \frac{1}{log_2(posRel+1)} & if\ posRel \geq 1 \end{cases}$$

where $posRel$ is the position of the relevant document in the ranking. It is worth noting that we consider a results window size of 10. Therefore, $posRel > 10$ implies the relevant document is not found.

In our tests, nDCG is high, i.e. near to one, when a relevant WSDL document is retrieved in the first or second position, but rapidly decreases as the relevant is ranked after the third position. For instance, nDCG is 1 when the relevant WSDL document is ranked in first place. However, nDCG is 0.63, 0.5, 0.3 and 0.29 when the document is ranked in the second, third, ninth and tenth positions respectively. Interestingly, nDCG value as regard to rank position is similar to the probability that a user selects a result in a specific position in a given search rank [1]. For instance, the probability of a user accessing the first position is near to one, but the probability of a user accessing the second position decreases to approximately 0.57.

On the other hand, Recall is an IR metric that computes the fraction of relevant documents in a data-set that were retrieved for a query by the registry. Formally:

$$Recall = \frac{RetRel}{Rel}$$

where $RetRel$ is the number of relevant documents retrieved by a query and Rel is the amount of relevant documents indexed by the registry. A variation of this metric is Recall-at-n, that only considers the first n ranked documents. This metric is particularly interesting for studying the first n ranked documents that registry users tend to prefer [1]. Again, since our queries have associated only one relevant Web Service, Recall-at-n can be zero or one. Similarly to the nDGC metric evaluation, we consider the average of Recall-at-n to analyze the results. Particularly, we study Recall-at-n with n ranging from 1 to 10, being this latter the default result window size of the Google search engine.

All reported nDCG and Recall values are averages of eighteen discoverability tests performed with each Noise data-set instance. Furthermore, from each of the 18 discoverability test pair of raw values <Third-party Data-Set, Relevant Data-Set> we performed Wilcoxon tests to verify whether the differences of discoverability measurements are statistically significant. Wilcoxon is a non-parametric test for comparing paired groups, or in our case, the nDCG values after retrieving WSDL documents generated with a third-party tool and with the Gapidt approach. Analogous statistical tests were performed for testing significance of obtained recall values. Our null (H_0) and alternative (H_1) hypotheses are:

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Registry	Original Axis2 (baseline)	Original Gapidt	**Gain (in %) of Gapidt**	$p-$value
WSQBE	0.684	0.739	**8.04**	0.0001964
Lucene4WSDL	0.720	0.748	**3.88**	0.0001964

Table 4: Original Axis2 and Original Gapidt: nDCG

- H_0 : The mean discoverability of WSDL documents resulting from the utilization of the Gapidt approach is the same with regard to the mean discoverability of WSDL documents resulting from the utilization of third-party tools.
- H_1 : The mean discoverability of WSDL documents resulting from the utilization of the Gapidt approach is not the same to that of WSDL documents resulting from the utilization of third-party tools.

We choose a significance level $\alpha = 0.01$ which means that the null hypothesis is rejected when p-values resulting from two-tailed Wilcoxon tests are less than 0.01.

4.3. Discoverability assessment of GMapper and third-party tools

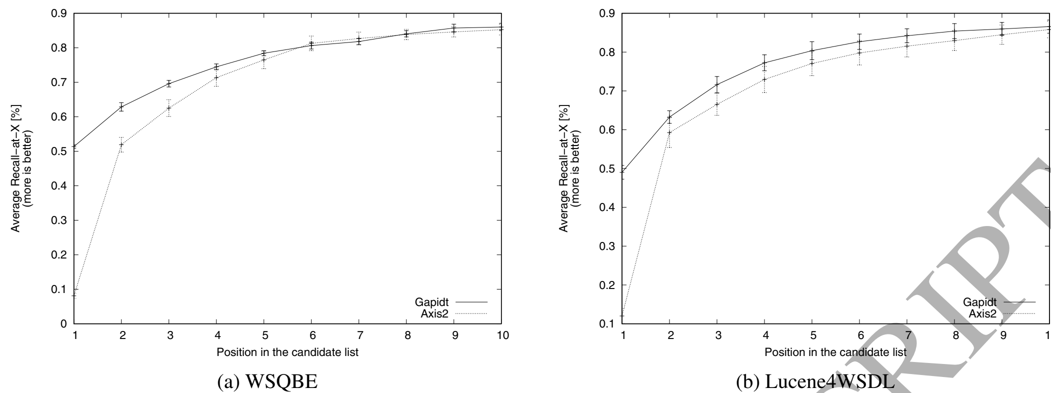
This section presents the discoverability evaluation of GMapper and third-party WSDL documents derived from the original Web Services source codes. Results of Axis2 versus GMapper are shown in Section 4.3.1, while EasyWSDL versus GMapper results are shown in Section 4.3.2. In both sections, we first analyze the discoverability through the nDCG metric and then through the Recall-at-n metric.

4.3.1. nDCG and Recall of Axis2 and GMapper

The discoverability assessment of Axis2 and GMapper WSDL documents is shown in Table 4, and Figure 8. Particularly, Table 4 shows the nDCG values for the two registries. In the Table, since all scenarios are favorable to Gapidt, the fourth column has been computed as $\frac{nDCG(Original\,Gapidt)-nDCG(Original\,Axis2)}{nDCG(Original\,Axis2)}$. nDCG values of GMapper WSDL documents, indicated as Original Gapidt, are higher than values for Axis2 for every Noise Data-Set. Specifically, with WSQBE the gain of GMapper over Axis2 is 8.04%, while with Lucene4WSDL it is 3.88%. Since p-values are less than the significance level chosen, the null hypotesis is rejected, which means that the discoverability achieved by the Gapidt approach is not equal to the discoverability of Axis2.

Figure 8 depicts the Recall-at-n values of Axis2 and GMapper WSDL documents and their standard deviation considering the different noise data-sets. Recall-at-one values for WSQBE (Figure 8a) are near to 0.5 for GMapper WSDL documents and 0.08 for Axis2 WSDL documents. This means that relevant GMapper WSDL documents at the first position were retrieved for around 50% of the queries, whereas relevant Axis2 documents were retrieved for only 8% of the queries. Since Recall-at-n is an accumulated function, values for the next positions of the ranking are ascending. In the sixth position the amount of relevant WSDL documents retrieved is approximately 0.8 for both tools (80%). With Lucene4WSDL (Figure 8b) Recall-at-n values behave very similar to the values for the WSQBE except for the first position, where Axis2 WSDL documents achieve 0.12 instead of 0.08, and that both curves join at the ninth position.

For both registries, Recall values obtained for different noises are coherent and there were no outliers. This is the reason why error bars or standard deviations are not long. Again, p-values of the Wilcoxon tests for Recall metric are 0.00019 which is less than the imposed threshold of 0.01, which allow us to reject the null hypotesis that states that the median discoverabilty difference of WSDL documents obtained with the Gapidt approach and WSDL documents obtained with Axis2 follows a distribution around zero. In terms of Recall-at-one, such discoverability difference is 40% in average in favor to the Gapidt approach.



(a) WSQBE (b) Lucene4WSDL

Figure 8: Original Axis2 and Original Gapidt: Recall-at-n

Registry	Original EasyWSDL (baseline)	Original Gapidt	**Gain (in %) of Gapidt**	$p-$value
WSQBE	0.691	0.743	**7.52**	0.0001964
Lucene4WSDL	0.696	0.762	**9.48**	0.0001964

Table 5: Original EasyWSDL and Original Gapidt: average nDCG

4.3.2. nDCG and Recall of EasyWSDL and GMapper

This section presents the discoverability assessment of WSDL documents generated via EasyWSDL and GMapper from the original code of Web Services. Table 5 shows that the nDCG values of GMapper WSDL documents are higher than those of EasyWSDL and result in a gain in favor to the Gapidt approach of 7.52% (WSQBE) and 9.48% (Lucene4WSDL). Again, since all scenarios are favorable to Gapidt, these two values have been computed as $\frac{nDCG(Original\,Gapidt)-nDCG(Original\,EasyWSDL)}{nDCG(Original\,EasyWSDL)}$. Moreover, these results pass the significance tests since p-values are less than 0.01. Figures 9 shows the Recall-at-n for EasyWSDL and GMapper WSDL documents. The curves are similar to those presented in Section 4.3.1, in the sense that GMapper values start over 0.5 reaching 0.9 at the tenth position of the ranking, while values of EasyWSDL start around 0.1 reaching also 0.9 at the tenth position. Then, nDCG and Recall-at-one values visually show a clear advantage of GMapper over EasyWSDL.

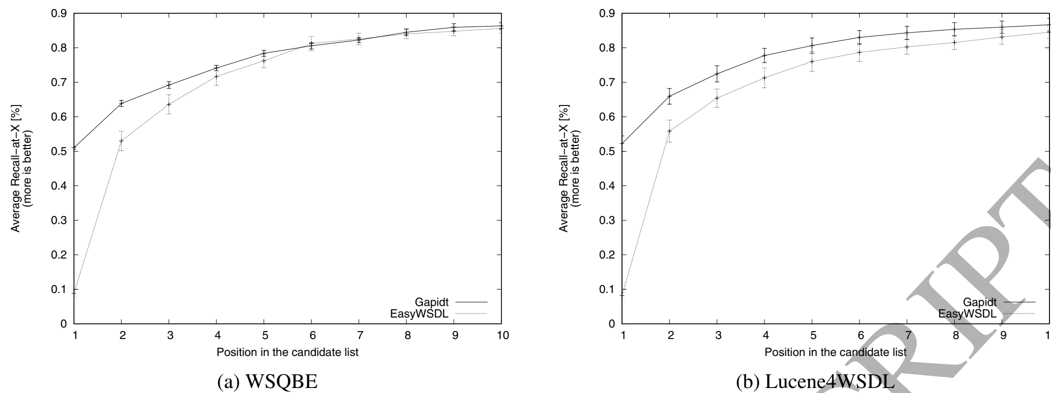
Like in the previous tests, the values obtained for every Noise data-set are coherent and do not present extreme values that affect our conclusions. Additionally, in all the experiments, we show that using Lucene4WSDL maintains GMapper over the other tool compared for all positions, while when using WSQBE, the tools begin to behave similarly from position 6 onwards. Again, the reported results are statistically significant since Wilcoxon tests for Recall-at-one threw p-values of 0.000195 and 0.000194 in experiments with WSQBE and Lucene4WSDL respectively. Again, given the significance level chosen (0.01), this allow us to reject the null hypotesis.

In summary, nDCG and Recall-at-one values show a clear advantage of GMapper over EasyWSDL. While the gain in terms of nDCG is at most 9.48%, the absolute difference in Recall-at-one is [42.21% - 44.02%].

4.4. Discoverability assessment of third-party tools and GAnalyzer + GMapper

This section assesses whether applying refactorings for detected anti-pattern root causes and using GMapper further increases WSDL document discoverability. This section focuses on the discoverability evaluation of the Gapidt approach combining GAnalyzer and GMapper (labeled as "refactored Gapidt") versus the above third-party tools. Although the tests include Recall values for the first ten positions of the ranking, we center the discussion in the Recall-at-one values because based on previous studies [1, 32], the

ACCEPTED MANUSCRIPT



(a) WSQBE (b) Lucene4WSDL

Figure 9: Original EasyWSDL and Original Gapidt: average Recall-at-n

first position is the one that exhibits the major interest to users. As a consequence of this and for simplicity, Recall-at-one values will be shown using tables rather than presenting Recall-at-n values with curves.

The subsections are organized as follows: Section 4.4.1 and Section 4.4.2 report the nDCG and Recall-at-one values of the Gapidt approach versus Axis2, while Section 4.4.3 and Section 4.4.4 report the values of the Gapidt approach versus EasyWSDL. We denote as *Individual Refactoring* the tests performed with a data-set obtained by spotting and removing (refactoring) only one root cause from the set supported by Gapidt, i.e. *AN* , *LCO* , *ILC* or *WT* . On the other hand, we denote as *Multiple Refactoring* the tests performed with a data-set that includes either all refactorings or all but *ILC* refactorings.

4.4.1. nDCG - Original Axis2 and Refactored Gapidt

Table 6 shows average nDCG values and difference percentages of discoverability tests for individual and multiple refactorings of Gapidt versus Axis2. The fifth column is computed as $abs(\frac{(Refactored\,Gapidt - Original\,Axis2}{Original\,Axis2}) * 100$, and an arrow head (▲) or (▼) precedes the values of the column when Gapidt obtained better performance than Axis2 or viceversa, respectively. Particularly, in 6 out of 8 individual refactorings tests the Gapidt approach outperforms the discoverability of WSDL documents generated with the third-party tool. The gains of Gapidt over Axis2 are present in all individual refactorings and range between [1.70, 14.20]%. The highest gains values, i.e., those over a 10%, are for the tests evaluating the effect of mitigating *WT* , meaning that WSDL documents whose source codes were modified to avoid generic types in the arguments and return values rank higher than Axis2 WSDL documents that were generated from the code with generic types. The advantage of the Gapidt approach is greater when using Lucene4WSDL (14.20%) than when using WSQBE (11.29%). Furthermore, irrespective of the registry used, the tests with middle-range gains, i.e., between 5% and 10%, are those related to refactoring for *AN* , while the lowest gains are associated to tests evaluating the effects of *LCO* and *ILC* refactorings. In the two last refactorings the gains are marginal but positive when using WSQBE. However, when using Lucene4WSDL, the performance differences are negative and in the range [6.16-10.12].

The refactorings for *ILC* were counter-productive for retrieving relevant documents. This occurs because the representation of the query differs more from the representation of a WSDL document with commented operations than a document with no commented operations. This also relates to the query detail level with which the search engine is consulted. As we mentioned in Section 4.1, the queries are generated based only on the service operation name, meaning that the extra text from comments cannot be fully matched against such simple queries.

When All Refactorings are applied, the Gapidt approach outperformed the baseline by 4.03% when using WSQBE, but losses by 3.29% when using Lucene4WSDL. When All Refactorings but *ILC* are applied, the gains of the Gapidt approach rise to 11.93% with WSQBE and 12.92% with LuceneWSDL. An observation from the multiple refactorings results is that refactorings for *ILC* hinders the gain achieved by other refactorings. This situation is in line with the results of refactoring for *ILC* and to a lesser extent refactoring for *LCO* tests, that do not show to be clearly beneficial to discoverability of WSDL documents.

ACCEPTED MANUSCRIPT

Refactoring	Registry	Original Axis2 (Baseline)	Refactored Gapidt	**Difference (in %) of Gapidt**	$p-$value
AN	WSQBE	0.703	0.753	▲7.11	0.0001964
	Lucene4WSDL	0.732	0.771	▲5.33	0.0001964
LCO	WSQBE	0.709	0.733	▲3.39	0.0001964
	Lucene4WSDL	0.730	0.685	▼6.16	0.0001964
ILC	WSQBE	0.706	0.718	▲1.70	0.0043370
	Lucene4WSDL	0.751	0.675	▼10.12	0.0001964
WT	WSQBE	0.762	0.848	**▲11.29**	0.0001960
	Lucene4WSDL	0.796	0.909	▲14.20	0.0001964
All Refactorings	WSQBE	0.720	0.749	▲4.03	0.0001964
	Lucene4WSDL	0.759	0.734	▼3.29	0.0001964
All Refactorings but *ILC*	WSQBE	0.696	0.779	**▲11.93**	0.0001964
	Lucene4WSDL	0.720	0.813	**▲12.92**	0.0001964

Table 6: Original Axis2 and Refactored Gapidt: Average nDCG

4.4.2. Recall-at-one - Original Axis2 and Refactored Gapidt

Table 7 shows the average Recall-at-one of Axis2 and Gapidt documents. The fifth column is computed as $abs(\frac{(Refactored\,Gapidt - Original\,Axis2}{Original\,Axis2}) * 100$, and an arrow head (▲) or (▼) precedes the recall values of the column when Gapidt performed better than Axis2 or viceversa, respectively. The discoverability effects of individual and multiple refactorings are similar to those observed with nDCG, i.e., the highest gains are obtained in refactoring for *WT* tests, followed by All Refactorings but *ILC* , refactoring for *AN* and refactoring for *LCO* tests –with the difference that, in this case, the latter report positive gains for both registries–. All Refactorings and refactoring for *ILC* tests comparatively show again less gains with WSQBE, and negative results with LuceneWSDL. Furthermore, in all tests, the p-values are under the chosen confidence level, which make the results statistically significant.

4.4.3. nDCG - Original EasyWSDL and Refactored Gapidt

Next, we analyze the nDCG values of EasyWSDL and refactored Gapidt WSDL documents.

Table 8 shows that nDCG differences of Gapidt are positive in 6 out of 8 tests of individual refactorings and in all multiple refactorings tests. These differences are computed as $abs(\frac{(Refactored\,Gapidt - Original\,EasyWSDL}{Original\,EasyWSDL}) *$ 100, and an arrow head (▲) or (▼) precedes the values of the fifth column when Gapidt obtained better performance than EasyWSDL or viceversa. Furthermore, several of these differences, particularly the tests that use LuceneWSDL, are notably higher than those reported in Table 6 for Gapidt versus Axis2. For instance, the gain in refactoring for *AN* tests increases from 5.33% to 9.35%. Furthermore, the loss in refactoring for *ILC* tests decreases from 10.12% to 7.02%, in All Refactorings tests from negative 3.29% to positive 1.24% and in All Refactorings but *ILC* from 12.92% to 17.49%. In all these cases, the p-values under the threshold allow us to conclude that the results are statistically significant except of refactoring for *LCO* test whose p-value is greater than 0.01. This means that the alternative hypothesis is rejected, so the null hypothesis (the discoverability in terms of nDCG of both variants do not differ) holds.

The test which use WSQBE do not present notable gains compared to Gapidt gain vs. Axis2. Only the refactoring for *WT* test reports a smaller gain than that of achieved by Gapidt vs. Axis2 for the same tests.

4.4.4. Recall-at-one analysis - Original EasyWSDL and Refactored Gapidt

Finally, we analyze the Recall-at-one values of tests conducted for EasyWSDL and Gapidt. Table 9 shows that the differences are, again, positive in favor to the Gapidt approach in the majority of discover-

ACCEPTED MANUSCRIPT

Refactoring	Registry	Original Axis2 (Baseline)	Refactored Gapidt	**Difference (in %) of Gapidt**	$p-$value
AN	WSQBE	0.112	0.492	▲339.29	0.0001960
	Lucene4WSDL	0.199	0.443	▲122.61	0.0001957
LCO	WSQBE	0.128	0.465	▲263.28	0.0001923
	Lucene4WSDL	0.141	0.432	▲206.38	0.0001943
ILC	WSQBE	0.237	0.346	▲45.99	0.0003270
	Lucene4WSDL	0.490	0.084	▼82.86	0.0001957
WT	WSQBE	0.011	0.760	▲6809.09	0.0001522
	Lucene4WSDL	0.080	0.672	▲740.00	0.0001759
All Refactorings	WSQBE	0.236	0.385	▲63.14	0.0001964
	Lucene4WSDL	0.458	0.180	▼60.70	0.0001926
All Refactorings but *ILC*	WSQBE	0.068	0.558	▲720.59	0.0001954
	Lucene4WSDL	0.148	0.513	▲246.62	0.0001957

Table 7: Original Axis2 and Refactored Gapidt: Average Recall-at-one

Refactoring	Registry	Original EasyWSDL (baseline)	Refactored Gapidt	**Difference (in %) of Gapidt**	$p-$value
AN	WSQBE	0.703	0.753	▲7.11	0.0001964
	Lucene4WSDL	0.706	0.772	▲9.35	0.0001964
LCO	WSQBE	0.698	0.734	▲5.16	0.0001964
	Lucene4WSDL	0.694	0.686	▼1.15	0.0936033
ILC	WSQBE	0.705	0.719	▲1.99	0.0049695
	Lucene4WSDL	0.726	0.675	▼7.02	0.0001964
WT	WSQBE	0.793	0.848	▲6.94	0.0001960
	Lucene4WSDL	0.789	0.909	▲15.21	0.0001960
All Refactorings	WSQBE	0.715	0.750	▲4.90	0.0001964
	Lucene4WSDL	0.725	0.734	▲1.24	0.0001964
All Refactorings but *ILC*	WSQBE	0.690	0.779	▲12.90	0.0001964
	Lucene4WSDL	0.692	0.813	▲17.49	0.0001964

Table 8: Original EasyWSDL and Refactored Gapidt: Average nDCG

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Refactoring	Registry	Original EasyWSDL (Baseline)	Refactored Gapidt	**Difference (in %) of Gapidt**	$p-$value
AN	WSQBE	0.109	0.496	▲355.05	0.0001957
	Lucene4WSDL	0.094	0.546	▲480.85	0.0001957
LCO	WSQBE	0.101	0.494	▲389.11	0.0001926
	Lucene4WSDL	0.089	0.477	▲435.96	0.0001827
ILC	WSQBE	0.214	0.377	▲76.17	0.0002332
	Lucene4WSDL	0.333	0.226	▼32.13	0.0001937
WT	WSQBE	0.034	0.743	▲2085.29	0.0001750
	Lucene4WSDL	0.076	0.679	▲793.42	0.0001769
All Refactorings	WSQBE	0.200	0.420	▲110.00	0.0001964
	Lucene4WSDL	0.317	0.311	▼1.89	0.0474516
All Refactorings but *ILC*	WSQBE	0.061	0.564	▲824.59	0.0001950
	Lucene4WSDL	0.037	0.627	▲1594.59	0.0001957

Table 9: Original EasyWSDL and Refactored Gapidt: average Recall-at-one

ability tests. The fifth column is computed as $abs(\frac{(RefactoredGapidt-OriginalAxis2}{OriginalAxis2}) * 100$, and an arrow head (▲) or (▼) precedes the values of the column when Gapidt obtained better Recall-at-one than Axis2 or viceversa, respectively. The negative differences affect the same tests reported in Table 7, i.e., refactoring for *ILC* and All Refactorings tests using LuceneWSDL. As pointed out in Section 4.4.3, the positive differences are higher when compared to the Recall-at-one values reported in Table 7. This suggests Gapidt Recall-at-one gains over EasyWSDL are in average greater than Gapidt Recall-at-one gains over Axis2 for tests using LuceneWSDL. The exception is the refactoring for *WT* test. Moreover, the gains of Gapidt vs. EasyWSDL tests that use WSQBE, are slightly greater in average w.r.t Gapidt vs. Axis2. All tests except the combination of All Refactoring with Lucene4WSDL throw p-values under the threshold, which means the discoverability is not statistically different.

4.4.5. Performance summary of GAnalyzer+GMapper

The results presented from Section 4.4.1 to Section 4.4.4 show the effects on discoverability of WSDL documents resulting from the application of individual and multiple refactorings to the original Web Service source code, and the usage of GMapper as WSDL generation tool. The tests contrast the discoverability of such documents with the discoverability of WSDL documents resulting from a traditional approach, i.e., without the application of refactorings and using traditional WSDL generation tools, for two different service discovery registries. Table 10 summarizes the average percentual difference of Gapidt w.r.t. its competing third-party tool discriminated by type of refactoring. It can be observed that the Gapidt approach increases the discoverability of WSDL documents irrespective of the third-party generation tool and the registry used in most cases.

According to the Table, the individual refactorings which always have positive impact on discoverability are *AN* , *LCO* and *WT* . Concerning to the refactoring for *ILC* , although its average impact on discoverability is negative for nDCG and positive in a marginal way for Recall-at-one, it presents an implicit benefit related to the readability of documents. This is, to the human reader it is naturally easier to understand the Web Service purpose when it has well-defined documentation than when it has not. Furthermore, the query generation mechanism explicitly ignores comments making the amount of text and semantic information in queries different from that in WSDL documents. This aims at using a similar criteria to that normally employed by users to express queries for inspecting service registries (i.e. supplying a few keywords [32]), but at the same time negatively affects the Web Service registry ability to match the

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Refactoring	Average nDCG difference	Average recall-at-one difference
AN	▲7.23	▲324.45
LCO	▲0.31	▲323.68
ILC	▼3.36	▲1.79
WT	▲11.91	▲2606.95
All Refactorings	▲1.72	▲27.64
All Refactorings but *ILC*	▲13.81	▲846.60

Table 10: Average discoverability differences introduced by the Gapidt approach discriminated by refactoring type

query with the relevant commented Web Service document.

When multiple refactorings were applied the negative effect of the refactoring for *ILC* did not help to achieve as considerable positive differences as when it was not considered. However, this does not represent an irremediable situation. To cope with this, IR registries could choose to ignore comments making discovery more effective for simple queries such as those using in the tests. The implementation of this would be straightforward: excluding comments when publishing/indexing documents and querying the stored specifications, but including them in the WSDL documents upon returning search results. Excluded comments could be maintained in separate documents and referenced from WSDL documents associated to them. In this way, high discovery is guaranteed, but relevant WSDL documents also include comments for users' perusal, which helps in increasing understandability.

The order in which several individual code refactorings are applied to a service code does not affect the resulting WSDL document because individual refactorings relate to disjoint concerns, i.e. each one acts on different source code elements. A change in the order of applying individual refactorings may, however, lead to a waste of analysis effort performed by the developer. This is the case of, for example, removing ambiguous names of a method that is later spotted as being non-cohesive. Clearly, removing the non-cohesive method first saves time as the name analysis is avoided.

4.5. Exploiting Gapidt: A service provider's and consumer's perspective

Section 4.3 presented tests that measured the discoverability of WSDL documents generated using GMapper only. The tests reveal that discoverability of GMapper WSDL documents outperforms that of documents obtained by third-party tools. In terms of nDCG, gains percentages are in the range [3.88% - 9.48%], whereas Recall-at-one gains percentages are in the range [36.78% - 44.30%]. This fact shows that WSDL generation practices, such as keeping original parameters names, avoidance of multiple definitions of the same data-type and avoidance of redundant port-types, contribute to improve discoverability. Moreover, in Section 4.4, we analyzed how using GAnalyzer plus GMapper affects discoverability. The results show that, when the individual refactoring for *WT* and All Refactorings but *ILC* are applied, the gains surpass those achieved by using GMapper only. The remaining refactorings proposed do not improve the gains achieved by using GMapper only but most of their gains are positive w.r.t. third-party tools. The tests of individual refactoring for *ILC* and those tests which involve All Refactorings present negative gains and the lowest gains, respectively, for both metrics. As pointed out, the solution to this would involve slightly modifying the internal mechanics of the registries, thus results in principle would be similar to that of scenarios including All Refactorings but *ILC* .

Despite not all individual refactorings used in conjunction with GMapper increase the discoverability achieved by the sole use of GMapper, all of them contribute to increase the understandability of WSDL documents. In SOC terms, *understandability* is a feature that can be associated to *usability*, which is a quality attribute defined in the ISO/IEC 25000 quality standard for standard software products. In fact, a recent survey analyzing 47 Web Service quality models [33] concluded that usability is the least explored software quality attribute in the literature. The studied refactorings indirectly represent practices that can be adopted to consider this important quality attribute. Therefore, the individual refactorings for *AN* and *ILC*

		OO refactorings				Gapidt refactorings			
		Original	Refactored	**Gain**	$p-$value	Original	Refactored	**Gain**	$p-$value
WSQBE	Axis2	0.676	0.731	**8.14%**	0.0001964	0.696	0.779	**11.93%**	0.0001964
	EasyWSDL	0.686	0.701	**2.19%**	0.0001964	0.690	0.784	**13.62%**	0.0001964
Lucene	Axis2	0.703	0.730	**3.84%**	0.0001964	0.720	0.813	**12.92%**	0.0001964
	EasyWSDL	0.668	0.710	**6.29%**	0.0001964	0.692	0.827	**19.51%**	0.0001964

Table 11: OO refactorings vs. Gapidt refactorings: average nDCG

and to a lesser extent *LCO* are, by excellence, refactorings which intuitively contribute to increase WSDL understandability. Even when their impact in discoverability is not as high as the refactoring for *WT* , if the developer aims at having good discoverability and understandability then the most balanced decision is to both fully refactoring their codes and using GMapper. In development scenarios where time is scarce and discoverability is preferred over understandability, just by using GMapper the resulting WSDL documents are not only more discoverable than documents generated with third-party tools, but also understandability is increased since the improved WSDL generation process further avoids some anti-patterns.

4.6. Ordiales Coscia et al.'s approach vs. Gapidt

This section presents a comparison of discoverability gains achieved by Ordiales Coscia et al.'s approach [31, 32] and Gapidt. The former is, to the best of our knowledge, the only alternative approach to Gapidt proposed in the literature to improve WSDL quality for *code-first* Web Services. We have analyzed both nDCG and Recall-at-one. Since Ordiales Coscia et al.'s approach (from now on "OO refactorings") includes service code refactorings but does not come with a new WSDL generation tool, we generated two OO refactorings data-sets, one using Axis2 and the other using EasyWSDL ("OO Refactorings -> Refactored" column in Tables 11 and 12). In both cases, Ordiales Coscia's approach was followed to refactor service codes. The discoverability was tested in experiments that include these data-sets, plus WSDL documents derived from the original, unrefactored Web Services source code ("Original" columns in Tables 11 and 12). For instance, valid tests combine, for all registries and all noise data-sets, the WSDL documents generated by Axis2 using the original service implementation data-set, and the WSDL documents generated by Axis2 for the OO-refactored service implementations. The same applies to the EasyWSDL generation tool. Since Axis2 and EasyWSDL disregard comments, we left the *ILC* anti-pattern out of the analysis, which means that we compare the discoverability gains of OO refactorings versus the discoverability gains of services refactored via Gapidt based on the All Refatorings but *ILC* refactoring (see column "Gapidt refactorings -> Refactored" in Tables 11 and 12).

Table 11 presents the nDCG results obtained. The two gain columns are computed as $abs(\frac{(Refactored-Original}{Original}) * 100$ using the original and refactored nDCG results from columns 3-4 and 7-8, respectively. From the Table it is derived that refactoring original services via Gapidt resulted in an average gain across the four tests of $\sim$14.49%, whereas refactoring original services via OO refactorings [31] resulted in an average gain of $\sim$5.11%. Note that p-values are less than the 0.01 threshold selected. Moreover, Table 12 presents the Recall-at-one results obtained for the OO refactorings tests and the Gapidt refactorings tests, where the gain columns were computed as in Table 11. In the same line of the nDCG results, the average percentual gains across the four tests obtained by Gapidt from refactoring original services compared to the percentual gains obtained by OO refactorings were higher: $\sim$846.59 and $\sim$213.56, respectively.

Again, all p-values under the threshold allow us to confirm the statistical significance of results.

Empirical results show that better discoverability levels are obtained by using Gapidt over the OO refactorings approach. This enforces the idea that applying text-oriented techniques to early remove anti-patterns outperformed the OO refactorings approach. To date, this hypothesis have not been conducted based on pure discoverability metrics, but by quantifying the number of anti-pattern occurrences in generated WSDL documents with the idea that such occurrences is an indirect indicator of service discoverability [32]. Then, this Section directly measured the effects in WSDL discoverability of Gapidt.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

		OO refactorings				Gapidt refactorings			
		Original	Refactored	**Gain**	$p-$value	Original	Refactored	**Gain**	$p-$value
WSQBE	Axis2	0.096	0.434	352.08%	0.0001954	0.068	0.558	**720.59%**	0.0001954
	EasyWSDL	0.163	0.376	130.67%	0.0001964	0.061	0.564	**824.59%**	0.0001950
Lucene	Axis2	0.172	0.381	121.51%	0.0001960	0.148	0.513	**246.62%**	0.0001957
	EasyWSDL	0.116	0.406	250.00%	0.0001957	0.037	0.627	**1594.59%**	0.0001957

Table 12: OO refactorings vs. Gapidt refactorings: average Recall-at-one

Although such OO refactorings are designed to take into account good API development practices to produce WSDL documents [19, 6, 17, 39, 13], the approach in [31] does not provide guidelines regarding the extent to which individual refactorings should be applied. Therefore, even when some refactorings are sound at a theoretical level, whether they lead to good APIs or not in practice depends heavily on how aggresively they are applied. For instance, a refactoring proposed is to "flat" complex data-types and replace them by primitive types (e.g. strings). What to do is ambiguous in presence of nested complex data-types. Then, in the above experiments, refactorings were applied strictly: all the data-types are simple data-types, and particularly complex types were replaced by ad-hoc character strings. Hence, there are still some API bad practices in the generated WSDL documents as a result of applying the refactorings suggested by the OO refactorings approach, compromising WSDL understandability.

5. Conclusions

We have assessed an approach called Gapidt aimed at removing discoverability anti-patterns from WSDL documents, which targets code-first Java Web Services. Gapidt is based on text mining techniques to spot the root causes of anti-patterns from service codes, code refactorings to remove them, and a WSDL generation tool that avoids those anti-patterns which depend on how codes are mapped to WSDL documents. Gapidt comes with a plug-in for the Eclipse IDE. All in all, we believe this assessment is crucial to bring Gapidt and its supporting tool to the industry.

We presented a thorough evaluation of how Gapidt can benefit discoverability of code-first Web Services. This work does not evaluate the Gapidt approach in a coarse-grained manner as in [29], but assesses Gapidt components and detection techniques separately. The first finding is that by only using GMapper the generated WSDL documents are more discoverable that when using two well-known third-party Java-to-WSDL generation tools (Axis2 and EasyWSDL). Furthermore, unlike GMapper, these tools disregard comments in the Java source code, mainly because they generate the WSDL documents from the compiled code of services and do not consider Javadocs as a potential source of comments.

The second experiment assessed how the different refactorings affect WSDL document discoverability. From Table 10 it can be concluded that applying each refactoring but the *ILC* led to more discoverable WSDL documents. The results for the *ILC* were negative when using Lucene4WSDL. We believe that this stems from the used experimental methodology: in order to avoid bias, we automatically generated the queries from the original WSDL documents contaning more anti-patterns. Therefore, the queries do not contain the terms that are present in the comments making the query vectors differ from the WSDL document vectors. Noticeable, although the queries are likely to benefit unrefactored WSDL documents, the refactored documents in most cases were more discoverable than the unrefactored ones. In fact, applying all the refactorings but the *ILC* also generates more discoverable WSDL documents, and by applying all the refactorings (i.e. including *ILC*) the results were slightly improved. However, the latter option generates more readable WSDL documents [14, 45] because produced documents contain comments that are essential in API definitions [24, 49]. As explained earlier, obtaining good discoverability while keeping comments in WSDL documents is possible by introducing slight technical modifications to the service registries.

In a third experiment, we have compared the discoverability of WSDL documents generated from service codes refactored with either Gapidt or the refactorings proposed by Ordiales Coscia et al. [31, 32].

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

We found that Gapidt outperforms the approach by Ordiales Coscia et al. It is worth noting that the latter bases on statistical correlations between OO source code metrics taken on the code implementing services and the amount of potential anti-pattern occurrences on generated WSDL documents. Hence, this approach probabilistically and indirectly links service code metrics to anti-patterns, while Gapidt takes a more direct approach to anti-pattern avoidance by spotting and removing anti-pattern root causes from service codes. In the future, however, an interesting research line is to study whether combining the various refactorings proposed by either approaches is viable and helps in further improving WSDL document quality.

We also plan to adapt our work in the context of good development practices for REST services and their implications. Indeed, REST services are rapidly gaining importance in the industry, but a recent study [13] has shown that REST developers do not follow good practices to define REST APIs. Furthermore, even when there is no standard for describing REST service APIs yet, some authors [27, 26, 23, 47] have proposed different approaches for REST service discovery and mash-up, which opens the door to further research in the area. For instance, in [47] a REST service discovery registry exploiting text processing techniques has been proposed. The registry is based on the Web Application Description Language –WADL, a specification language analogous to WSDL–, and thus good practices for producing WADL documents could be studied, as well as their impact on WADL discoverability and understandability. Last but not least, other future tasks relate to the tool implementing the approach itself, so it can be readily used by practitioners in the industry. This includes support for various Java IDE apart from Eclipse, and better integration with Java annotations. Particularly, the well-known JAXB specification defines standard annotations for building code-first Java Web Services while specifying in the code several aspects such as exposed business objects and data marshalling/unmarshalling schemes. At present, Gapidt supports importing a (JAXB) pre-generated schema instead of defining the schema within the WSDL document. Thus, JAXB schemas can be used without much difficulty. Moreover, since Gapidt is a service code refactoring and WSDL document generation tool, Gapidt is not concerned with XML-Java classes marshalling and unmarshalling, which is the other key functionality of JAXB. However, we will improve the integration of Gapidt with JAXB.

Acknowledgments

We acknowledge the financial support by ANPCyT through grant PICT-2012-0045 and PICT-2013-0464. The first and second authors acknowledge their Ph.D. scholarship granted by the CONICET.

References

[1] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning user interaction models for predicting web search result preferences. In *SIGIR '06, Seattle, Washington, USA*, pages 3–10. ACM, 2006.

[2] Jehad Al Dallal. The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities. *Journal of Systems and Software*, 85(5):1042 – 1057, 2012.

[3] E. Al-Masri and Q. Mahmoud. QoS-based discovery and ranking of web services. In *ICCCN 2007, Honolulu, Hawaii, USA*, pages 529–534. IEEE, August 2007.

[4] Eric Allen and Robert Cartwright. Safe instantiation in generic java. *Science of Computer Programming*, 59(1-2):26–37, 2006.

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

[6] Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Jack, and Brad Myers. Usability challenges for enterprise service-oriented architecture APIs. In *IEEE VL/HCC 2008*, pages 193–196, 2008.

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

ACCEPTED MANUSCRIPT

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

[9] Heung Seok Chae, Yong Rae Kwon, and Doo Hwan Bae. A cohesion measure for object-oriented classes. *Software: Practice and Experience*, 30(12):1405–1431, 2000.

[10] Christopher D. Manning Dan Klein. Accurate unlexicalized parsing. In *41st Meeting of the Association for Computational Linguistics*, 2003.

[11] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[12] Bill Dudney, Joseph Krozak, Kevin Wittkopf, Stephen Asbury, and David Osborne. *J2EE Antipatterns*. John Wiley; Sons, Inc, 8 2003.

[13] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, February 2015.

[14] Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32, 2005.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[16] Martin Garriga, Cristian Mateos, Andres Flores, Alejandra Cechich, and Alejandro Zunino. Restful service composition at a glance: A survey. *Journal of Network and Computer Applications*, 60:32–53, 2016.

[17] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. Methods towards api usability: A structural analysis of usability problem categories. In *Human-Centered Software Engineering*, volume 7623 of *LNCS*, pages 164–180. Springer, 2012.

[18] Samer Hanna. An approach to detect web services vague datatype specifications to enhance understandability. *Int. J. Web Eng. Technol.*, 11(1):68–103, April 2016.

[19] Michi Henning. API design matters. *Communications of the ACM*, 52(5):46–56, May 2009.

[20] Andreas Heß, Eddie Johnston, and Nicholas Kushmerick. ASSAM: A tool for semi-automatically annotating semantic Web Services. In *International Semantic Web Conference*, volume 3298 of *LNCS*, pages 320–334, November 2004.

[21] Einar Høst and Bjarte Østvold. Debugging method names. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 294–317. Springer, 2009.

[22] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems*, 20(4):422–446, October 2002.

[23] Davis John and M. Rajasree. A framework for the description, discovery and composition of RESTful semantic Web Services. In *2nd International Conference on Computational Science, Engineering and Information Technology, Coimbatore UNK, India*, pages 88–93. ACM, 2012.

[24] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: The javadocminer. In *Natural Language Processing and Information Systems*, volume 6177 of *LNCS*, pages 68–79. Springer, 2010.

[25] Peter Kincaid, Robert Fishburne, Richard Rogers, and Brad Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, DTIC Document, 1975.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[26] J. Kopecky, K. Gomadam, and T. Vitvar. hrests: An html microformat for describing RESTful Web Services. In *WI-IAT '08*, volume 1, pages 619–625, Dec 2008.

[27] J. Lathem, K. Gomadam, and A. Sheth. SA-REST and (s)mashups : Adding semantics to RESTful services. In *International Conference on Semantic Computing*, pages 469–476, Sept 2007.

[28] Dawn Lawrie, Henry Feild, and David Binkley. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.

[29] Cristian Mateos, Juan Rodriguez, and Alejandro Zunino. A tool to improve code-first web services discoverability through text mining techniques. *Software: Practice and Experience*, 45(7):925–948, 2015.

[30] George Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, November 1995.

[31] José Luis Ordiales Coscia, Cristian Mateos, Marco Crasso, and Alejandro Zunino. Anti-pattern free code-first web services for state-of-the-art java wsdl generation tools. *International Journal of Web and Grid Services*, 9(2):107–126, 2013.

[32] José Luis Ordiales Coscia, Cristian Mateos, Marco Crasso, and Alejandro Zunino. Refactoring code-first web services for early avoiding {WSDL} anti-patterns: Approach and comprehensive assessment. *Science of Computer Programming*, 89, Part C(0):374–407, 2014.

[33] Marc Oriol, Jordi Marco, and Xavier Franch. Quality models for web services: A systematic mapping. *Information and Software Technology*, 56(10):1167 – 1182, 2014.

[34] A. Ouni, M. Kessentini, K. Inoue, and M. O Cinneide. Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, PP(99):1–1, 2017.

[35] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. Web service antipatterns detection using genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*, pages 1351–1358, New York, NY, USA, 2015. ACM.

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

[37] J. Pasley. Avoid XML schema wildcards for Web Service interfaces. *IEEE Internet Computing*, 10(3):72–79, 2006.

[38] John Kelly Villota Pismag. Prediction of web service antipatterns using machine learning. Master's thesis, University of Michigan-Dearborn, 04 2017.

[39] Girish Maskeri Rama and Avinash Kak. Some structural measures of api usability. *Software: Practice and Experience*, 45(1):75–110, 2015.

[40] Alan De Renzis, Martin Garriga, Andres Flores, Alejandra Cechich, Cristian Mateos, and Alejandro Zunino. A domain independent readability metric for web service descriptions. *Computer Standards & Interfaces*, 50:124–141, 2017.

[41] Guillermo Rodríguez, Álvaro Soria, Alfredo Teyseyre, Luis Berdun, and Marcelo Campo. *Unsupervised Learning for Detecting Refactoring Opportunities in Service-Oriented Applications*, pages 335–342. Springer International Publishing, Cham, 2016.

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

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[43] Juan Rodriguez, Marco Crasso, Cristian Mateos, and Alejandro Zunino. Best practices for describing, consuming, and discovering web services: A comprehensive toolset. *Software: Practice and Experience*, 43(6):613–639, 2013.

[44] Juan Rodriguez, Marco Crasso, and Alejandro Zunino. An approach for web service discoverability anti-patterns detection. *Journal of Web Engineering*, 12(1–2):131–158, 2013.

[45] Juan Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.

[46] Juan Rodriguez, Cristian Mateos, and Alejandro Zunino. Assisting developers to build high-quality code-first web service apis. *Journal of Web Engineering*, 14(3&4):251–285, 2015.

[47] Juan Rodriguez, Alejandro Zunino, Cristian Mateos, Felix Segura, and Emmanuel Rodriguez. Improving rest service discovery with unsupervised learning techniques. In *9th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 97–104, July 2015.

[48] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In *IWPSE '07, Dubrovnik, Croatia*, pages 4–10. ACM, 2007.

[49] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension*, pages 83–92, 2013.

[50] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky. Identification of web service refactoring opportunities as a multi-objective problem. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 586–593, June 2016.

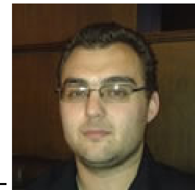
[51] Chen Wu. Wsdl term tokenization methods for ir-style web services discovery. *Science of Computer Programming*, 77(3):355 – 374, 2012.

[52] Guobing Zou, Yanglan Gan, Yixin Chen, and Bofeng Zhang. Dynamic composition of Web Services using efficient planners in large-scale service repository. *Knowledge-Based Systems*, 62:98–112, 2014.

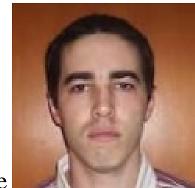
ACCEPTED MANUSCRIPT

Matías Hirsch holds a BSc. in Systems Engineering from the Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN). He is a teaching assistant at the UNICEN. He is currently pursuing his Ph.D. in Computer Science working under the supervision of Cristian Mateos and Alejandro Zunino. Contact him at matias.hirsch@isistan.unicen.edu.ar.

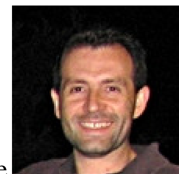
Ana Rodriguez holds a BSc. in Systems Engineering from the UNICEN. She is a teaching assistant at the UNICEN. She is currently pursuing her Ph.D. in Computer Science working under the supervision of Alejandro Zunino and Cristian Mateos. Contact her at ana.rodriguez@isistan.unicen.edu.ar.



Juan Manuel Rodriguez holds a Ph.D. degree in Computer Science from the UNICEN. He is a teaching assistant at the UNICEN and researcher at the CONICET. His research interests include Web services, mobile devices, grid computing, and service-oriented grids. Contact him at `www.exa.unicen.edu.ar/~jmrodri` and jmrodri@exa.unicen.edu.ar.



Cristian Mateos has an MSc. and a Ph.D. in Computer Science from UNICEN. He is an adjunct professor at the UNICEN and researcher at the CONICET. He is interested in parallel and distributed programming, middlewares, and mobile/service-oriented computing. Contact him at `www.exa.unicen.edu.ar/~cmateos` or cristian.mateos@isistan.unicen.edu.ar.



Alejandro Zunino has an MSc. and a Ph.D. in Computer Science from UNICEN. He is an adjunct professor at the UNICEN and researcher at the CONICET. His research areas include grid computing, service-oriented computing, Semantic Web services, and computer security. Contact him at `www.exa.unicen.edu.ar/~azunino` or alejandro.zunino@isistan.unicen.edu.ar.

ACCEPTED MANUSCRIPT