Science of Computer Programming 89 (2014) 374–407

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Refactoring code-first Web Services for early avoiding WSDL anti-patterns: Approach and comprehensive assessment

José Luis Ordiales Coscia, Cristian Mateos *,[1], Marco Crasso, Alejandro Zunino

ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil B7001BBO, Argentina

HIGHLIGHTS

- An approach to avoid WSDL anti-patterns in code-first, Java-based Web Services.
- An evaluation with modern Java to WSDL tools and registries, plus real Web Services.
- Results showing that code-first Web Services are more understandable and retrievable.
- Guidelines for using the approach in the industry through a cost-benefit analysis.

ARTICLE INFO

Article history:
Received 23 April 2013
Received in revised form 20 November 2013
Accepted 31 March 2014
Available online 13 April 2014

Keywords:
Web services
Code-first
WSDL anti-patterns
Service understandability
Service retrievability

ABSTRACT

Previous research of our own [34] has shown that by avoiding certain bad specification practices, or WSDL anti-patterns, contract-first Web Service descriptions expressed in WSDL can be greatly improved in terms of understandability and retrievability. The former means the capability of a human discoverer to effectively reason about a Web Service functionality just by inspecting its associated WSDL description. The latter means correctly retrieving a relevant Web Service by a syntactic service registry upon a meaningful user's query. However, code-first service construction dominates in the industry due to its simplicity. This paper proposes an approach to avoid WSDL anti-patterns in code-first Web Services. We also evaluate the approach in terms of services understandability and retrievability, deeply discuss the experimental results, and delineate some guidelines to help code-first Web Service developers in dealing with the trade-offs that arise between these two dimensions. Certainly, our approach allows services to be more understandable, due to anti-pattern remotion, and retrievable as measured by classical Information Retrieval metrics.

1. Introduction

Web Services refers to a set of standard technologies for materializing the Service-Oriented Computing paradigm (or SOC for short) [26,14]. Web Services allow service providers to expose their services using well-known interoperable Web protocols, such as HTTP or SOAP. In this context, services are offered using sets of atomic operations that are described using the Web Service Description Language (WSDL). Plainly, WSDL bases on XML and allows publishers to describe the functionality of their Web Services as a set of abstract operations with inputs and outputs defined in the XML Schema

* Corresponding author. Tel.: +54 249 4439682x35; fax: +54 249 4439681.
E-mail addresses: jlordiales@gmail.com (J.L. Ordiales Coscia), cmateos@conicet.gov.ar (C. Mateos), mcrasso@gmail.com (M. Crasso), azunino@conicet.gov.ar (A. Zunino).
[1] Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

http://dx.doi.org/10.1016/j.scico.2014.03.015
0167-6423/

Definition (XSD) language, and to specify the associated binding information so that clients can actually consume the offered operations. The goal of WSDL and XSD is that service consumers can reason about the functionality of a Web Service without knowing service implementation details and by only looking at its associated WSDL document. Intuitively, this can be achieved provided the document is *understandable*, i.e., it does not suffer from specification problems (e.g., lacking comments) that might obscure the purpose of the service when inspected by a user.

WSDL documents are useful on the other hand for indexing services in *syntactic registries* so that service consumers can look for the services they need. A syntactic registry is simply a repository of WSDL documents with functionality to match a list of keywords – obtained from either natural language or structured user queries – against the main index built from these documents. Moreover, indexes are built by extracting keywords from published WSDL documents, and then modeling the extracted information as inverted indexes or vector spaces [13]. Then, the generated models are employed for retrieving relevant WSDL documents for a given keyword-based query. These approaches are strongly inspired by classic Information Retrieval (IR) techniques, such as word sense disambiguation, stop-words removal, and stemming. Despite their pragmatism, feasibility, and low cost of adoption, their effectiveness decreases when dealing with poorly specified WSDL documents [7,27]. Indeed, it has been shown in practice that publicly available WSDL documents do not contain useful pieces of information (e.g., ambiguous names) making Web Service syntactic registries ineffective [10].

The process of discovering Web Services, as suggested, encompasses two related tasks, automatically matching a query against potential candidate services in a syntactic registry, and then manually inspecting resulting candidates to select the desired result. For this process to be effective, service providers must provide meaningful WSDL documents upon developing services [10]. For the case of *contract-first* Web Services, in which WSDL documents are manually specified before implementing services, it is known that by considering a catalog of bad WSDL practices – i.e., WSDL anti-patterns – the "retrievability" and "understandability" of WSDL documents is greatly improved [34]. These anti-patterns represent bad practices when specifying WSDL documents that relate to the way port-types, operations and messages are structured and modeled.

The most popular approach to build Web Services in the industry is code-first, whereby developers first implement a service and then generate the corresponding WSDL by automatically extracting and deriving the interface from the implemented code. In other words, WSDL documents are not manually built by developers but are automatically derived via language-dependent tools such as Axis' Java2WSDL [38]. This approach prevents in principle the developer from manually specifying WSDL documents so as to be free from WSDL anti-patterns. We have however found that there is a high statistical correlation between established Object-Oriented (OO) metrics taken from the source code implementing services and the occurrences of anti-patterns in the generated WSDL documents [24]. This correlation enables the use of simple source code refactorings to modify the values of OO metrics, and in this way indirectly reduce the occurrences of anti-patterns in WSDL documents. Moreover, in [20], preliminary experiments in the context of Java-based Web Services and the Axis' Java2WSDL tool have shown that these refactorings are useful for improving the "retrievability" of Web Services. The experiments were performed by populating a registry with original and refactored services and measuring retrievability by using several classical IR metrics.

The purpose of this paper is to refine the results in [34] and [20] in order to further extend and generalize our findings, respectively. It is worth noting that our research is still focused on Web Services implemented in Java, a very popular language in back-end development. Concretely, our contributions are the following:

- We deeply study the effects of removing WSDL anti-patterns in WSDL understandability and retrievability for the case of Web Service descriptions obtained through code-first, to complement similar research already done in the context of contract-first Web Services [34],
- In terms of the retrievability dimension and since the WSDL generation tool has an acknowledgeable incidence in this respect [24], unlike [20], we report on experiments by using more generation tools apart from Java2WSDL to assess retrievability. The extra tools employed include EasyWSDL [25], Java2WS [4] and WSProvide [17]. Results are measured in terms of Precision-at-*n*, Recall and Normalized Discounted Cumulative Gain (NDCG),
- Unlike [20], we include more realistic discovery scenarios by considering registries with a larger number of Web Services. This is done by using real WSDL data-sets gathered from public repositories, which are published in the registries together with the documents obtained from the code-first services under study, and
- We qualitatively analyze and position the effort required by each source code refactoring versus the potential gains in terms of both WSDL document retrievability and understandability. Retrievability measures how effective service registries are at finding and ranking relevant WSDL descriptions, while understandability refers to the effort required by developers to determine what operations from a Web Service they need to use and how they should invoke it. We believe that the generalized nature of our analysis might guide service developers to select which refactoring to apply according to their WSDL quality goals.

The rest of the paper is organized as follows. Section 2 discusses related work on improving WSDL documents according to the dimensions introduced above. Then, Section 3 presents the necessary concepts to understand the underpinnings of our work, including the approach to correlate OO metrics and WSDL anti-patterns. After that, Section 4 presents a comprehensive qualitative and quantitative analysis of the impact of service refactoring in the retrievability and understandability of WSDL documents. Finally, Section 5 concludes the paper.

2. Related works

Given the critical role that WSDL descriptions play on the successful use of Web Services, a substantial amount of research efforts have focused on common mistakes made by developers when specifying these documents and possible solutions to them. The following sections will discuss past research on common bad practices present in WSDL documents.

2.1. Blake and Nowlan

In [7] the authors focus on the developers' tendencies to use a standard nomenclature or predictable phrases during the specification of their WSDL documents. The authors set forward the hypothesis that the detection of these "naming tendencies" can help service retrieval with the use of a syntactic approach. The names of message parts in 596 WSDL documents, which were gathered from Internet repositories such as *Woogle*, *Binding-Point*, *Salcentral*,[2] *XMLMethods*,[3] and *WebServiceList*[4] were analyzed. They divide in/out messages into crumbled parts. Afterward, the name of each part is compared against the rest of the part names.

The authors identified several naming tendencies in the observed service parts. The first tendency showed that developers use similar phrases within part names. This means that despite the 596 Web Services had been developed by different developers and organizations, similar names were used to represent semantically equivalent concepts. Concretely, out of the 31 807 WSDL parts extracted only 25% were unique. For example, the authors found that a message part standing for a user's name is called "*name*", "*lname*", "*user_name*" or "*first_name*" [7]. Moreover, the second identified tendency showed that most of the analyzed Web Services, which were active and deployed on real registries, contained non-representative names. The authors define a name to be non-representative if it contains less than 3 characters or terms that are not related to the part they describe, such as "in0", "arg1" or "foo". Finally, the authors identified the developers' tendency of shortening phrases into abbreviations. For instance, using "*bldg*" instead of "*building*" or "*cntry*" instead of "*country*". Then, the authors supply a syntactic registry with heuristics designed for dealing with the identified naming tendencies. As a result, they empirically showed that the retrieval effectiveness of a syntactic registry can be improved by enhancing its underlying matching approach to deal with these tendencies.

2.2. Pasley

In [27] analyzed the impact of using "XSD wild-cards", when defining data-types, on the maintainability and retrievability of Web Services. A wild-card is a special XSD constructor, e.g., *xsd:any* and *xsd:anyAttribute*, which allows developers to leave one or more parts of an XML structure undefined. These constructs were defined by the W3C[5] to provide an extension point and allow an XML Schema to remain unchanged when the underlying format of the data is modified. This is equivalent to the concept of base classes on most OO languages such as Java or Ruby. However, wildcards introduce unwanted secondary effects, such as extra complexity to understand service descriptions.

One detected reason for using such XSD constructors is to minimize the effort involved in modifying a service when it evolves, while assuring that consumers bound to old versions of the service will be able to correctly invoke and process the operations defined in its new version [27]. The idea is to include extension points in a WSDL document and postponing the definition of some of its constituent parts. The author asserts that the main drawback of using such an extensibility mechanism is defining "vague interface contracts". A WSDL document is said to be vague when it defines at least one of its messages by means of XSD wildcards, because it is not possible to express which extensions are supported. Imagine an operation that defines its output using an *xsd:any*-based element, which means that the output can be any valid XML, then the output definition does not allow discoverers to exactly infer what the service response will be like. Therefore, the author presents a different versioning strategy for WSDL documents, which removes XSD wildcards from data-types, and facilitates Web Services forward compatibility at the same time.

2.3. Beaton et al.

In [6] the authors describe many difficulties that six students of a SOC course encountered while developing a large Web Services-based application. Some of the identified difficulties relate to understanding third-party Web Services. Specifically, the authors show that unclear "control parameters" within data structures and long identifier names make a Web Service harder to be understood [6]. A message associated with a control parameter, besides carrying data objects, includes miscellaneous objects that are usually not well documented. As this kind of objects tends to control the execution flow of service consumers, the authors refer to them as "control parameters". Control parameters are frequently used to inform about errors that occur during the execution of an operation. Moreover, the authors observe that all six users either misread names or were confused by them, and all complained about the length of the names.

[2] Unfortunately, Woogle, Binding-Point and Salcentral repositories are no longer available.
[3] http://www.xmethods.net/.
[4] http://www.webservicelist.com/.
[5] http://www.w3.org/.

For their experiments, the authors gave the six students the WSDL descriptions of four different Web Services that had to be used to implement an order tracking system. The only imposed restrictions were to use the Web Services on an established order, namely "Find Customer", "Find Supplier", "Find Product" and "Create Sales Order". Finally, the authors performed several tests using the Think Aloud protocol [29]. This technique involves participants thinking out loud while they perform the tasks assigned to them. The participants are asked to talk about everything they see, hear, feel or think, while the observers take note of everything without trying to personally interpret the data. This allows the observers to have a really detailed description of the whole work process and not just the final results.

The results of the study showed that the students encountered a set of difficulties while completing the assigned task. The authors then classified these difficulties based on the error that caused them:

- Assembly Error ("Some Assembly Required"): this error occurred when the developer was not able to correctly create and combine some or all the messages required. For example, on the "Find Customer" service a *Customer* object had to be sent containing a *Common* object. The lack of help on the client code generated from the WSDL document caused several "Null Reference" exceptions.
- Exclusion Error ("Batteries Not Included"): additional to the required objects that transport the information between the client and the service, the students sometimes had to include optional objects such as logs. Given that most of these objects were not documented, the developers usually ignored them.
- Specification Error ("Do Not Eat Silicon Gel"): some objects did not need to be explicitly created and assigned in order to work correctly. The students made several mistakes while incorrectly combining the different fields of the data structures. This was the most a time consuming error to fix.
- Structure Error: all the students were confused when facing the creation of the basic data structures, showing difficulties trying to determine the meaning of each field.
- Default Error: three out of the six students were not sure if an object needed to be created or just assigned some default value. They all tried to guess the required values, leading them to the Specification Errors issue.
- Foundation Error: all the students were confused by the stub code generated from the WSDL file.
- Naming Error: all the six students were confused by the names of the different identifiers. They all complained about the length of these names and incurred on the "Specification Error" mistake by confusing long names that were only different by a subtle word in the middle of them.
- Consistency Error: two students tried to copy and paste the code for the "Find Customer" and use it for "Find Customer" before noticing that the code was quite different. This was due to the fact that the data structures for both of them look very similar while using two completely different design patterns.

After identifying all the previous issues, the authors propose a set of recommendations that try to eliminate, or at least mitigate, some of the difficulties encountered by the users. For example, to prevent *Assembly Errors* the authors suggest to enable stub client code generated from WSDL documents to automatically instantiate the required objects with default values to indicate that these objects are uninitialized. This also implies that services should tolerate objects supplied as parameters with fields that are uninitialized or empty.

2.4. Rodriguez et al.

In [34] the authors analyzed a set of 391 public WSDL documents and conformed a catalog of bad practices, or WSDL anti-patterns, usually present on the description of Web Services. It was found that the total number of occurrences of each anti-pattern in the analyzed data-set, in terms of comment-related problems, affects up to 82% of the data-set. Similarly, even though the definition of the data schema within the WSDL document makes its reuse harder in other documents, 70% of the services from the data-set define their data-types in embedded form. As a result of this experiment, the authors proposed a set of refactoring operations that can be applied to the WSDL documents to remove the presence of these bad practices.

In [34] the same authors measured the importance of WSDL anti-patterns in service retrievability by manually removing anti-patterns from a data-set of around 400 WSDL documents and comparing the retrieval effectiveness of different syntactic registries when using the original WSDL documents and the improved ones, i.e., the WSDL documents that have been refactored according to the anti-pattern solutions. The methodology used for this experiment involved the publication of the two data-sets on each service registry, followed by a set of queries performed on each one to measure the relevance of the retrieved WSDL documents and the efficiency of the process. The goal of the experiment was to show how the removal of the anti-patterns identified had a positive impact on service retrieval.

The authors concluded that, regardless of the service registry used, the results related to the improved data-sets surpassed those achieved by using the original data-set. This provides evidence that suggests that the improvements are explained by the removal of the anti-patterns rather than the incidence of the underlying retrieval mechanism. The authors then state that "There is no silver bullet to guarantee that potential consumers of a Web Service will effectively retrieve, understand and access it. However, we have empirically shown that a WSDL document can be improved to simultaneously address these issues by following certain steps. By removing anti-patterns, the proposed guide allows service publishers

to improve the cohesion, reusability, readability, and discoverability of their service descriptions provided they are able to control their WSDL documents".

As a consequence of these findings, and to guide developers in quantifying the extent to which a given WSDL is affected by anti-patterns, an automatic anti-pattern detection tool has been developed [32]. Basically, the tool offers and implements simple WSDL metrics to count anti-pattern occurrences via text and XML mining techniques. Given a WSDL document, the tool returns a table with anti-patterns occurrences as computed by the WSDL metrics. It is worth noting that this tool was used to compute the values of the independent variables (i.e., anti-patterns) of the statistical correlation model presented in this paper, as we will describe later. For details on the theoretical and algorithmic underpinnings regarding anti-pattern detection in this otol, please refer to [32,33].

2.5. Discussion

All the studies previously mentioned focus on improving the quality of WSDL documents from a contract-first point of view. This means that they assume that developers manually specify their descriptions before implementing services. However, this is not the case in the software industry [24,20]. Speed and ease of development cause developers to favor code-first development with the use of automatic WSDL generation tools that take the source code of a service as input and generate a WSDL document for that service as output. Those documents usually suffer from most of the anti-patterns described in [34] as a result of bad coding practices, or deficient generation processes by the employed tools. That is, even though the generated description is functionally valid, the generation tool looses some relevant information in the process while generating redundant and unnecessary data. As a result, our research efforts [20] have been directed to address the quality of Web Services from a code-first perspective, where OO metrics are used as indicators for the number of occurrences of anti-patterns. In this way the negative effects of the WSDL anti-patterns can be minimized at an early stage of Web Service development by applying certain refactoring operations driven by these metrics.

3. Early revising code-first WSDL documents: background and approach

The studies presented in [34,10] identify recurrent bad practices, called WSDL anti-patterns, that take place in public WSDL documents and jeopardize WSDL discoverability in general. Particularly, these practices have been proved to negatively impact both syntactic registries retrieval effectiveness and WSDL legibility as perceived by human discoverers. Therefore, the studies propose guidelines to remedy the identified problems as a catalog of WSDL anti-patterns. Table 1 shows the core sub-set of the catalog studied in this paper. Broadly, all anti-patterns are classified in three categories, namely high-level service interface specification problems, comments and identifiers problems, and service message exchange problems.

Moreover, these guidelines consist of refactoring actions that should be applied over problematic WSDL documents. Then, given a WSDL document having anti-patterns, its publisher can methodically modify it until all anti-patterns have been removed. Only after this is done, publishers should concentrate on implementing the logic of the service [34]. As such, these refactoring actions can be applied when following contract-first methodology to build services only. However, contract-first is not popular among developers because it requires more effort than code-first. Publishers must master the WSDL specification and the XSD data-type language. On the other hand, code-first simply generates WSDLs from existing service code. In the end, publishers focus on developing and maintaining services implementations in any programming language, while delegating WSDL generation to specialized tools during service deployment [24].

Therefore, in [24], an approach to avoid or at least reduce WSDL anti-patterns in generated WSDL documents when using the code-first methodology was proposed. When using code-first, there are some relationships between the source code of a service implementation and its generated WSDL document. The approach is based on the fact that WSDL anti-patterns can be predicted by taking a set of OO metrics on services implementations, i.e., WSDL documents can be "early" revised. These metrics are depicted in Table 2. This rationale assumes that a typical code-first tool performs a mapping $T : C \rightarrow W$, being $C = \{M(I_0, R_0), ..., M_N(I_N, R_N)\}$ the frontend class implementing a service, and $W = \{O_0(I_0, R_0), ..., O_N(I_N, R_N)\}$ the WSDL document describing the service containing a *port-type* for the service implementation class and as many *operations* O as public methods M are defined in class C. Moreover, each *operation* of W is associated with one input *message* I and a return *message* R, while each *message* conveys an XSD type that models the parameters of the corresponding class method. Code-first tools such as WSDL.exe, Java2WSDL, and gSOAP are based on a mapping T for generating WSDL documents from C#, Java and C++, respectively, though each tool implements T in a particular manner mostly because of the different characteristics of the targeted programming languages. Then, the measurable properties of services implementations may influence resulting WSDL documents.

To test the statistical correlation among OO metrics and the WSDL anti-patterns, an exploratory approach was used, whose goal was to evaluate the feasibility of avoiding WSDL anti-patterns by taking into account Object-Oriented (OO) metrics from the code implementing services. It was found that the statistical hypotheses listed below present statistical significant relationships between the two kinds of variables of the associated correlation model, given by OO metrics (independent variables) and anti-patterns (dependent variables):

Table 1
The core sub-set of the WSDL anti-patterns.

Anti-pattern	Category	Occurs when	Affects retrievability, since...
Enclosed data model	High-level service interface specification	Data-type definitions used for exchanging information are placed in WSDL documents rather than in separate XSD documents.	Syntactic registries extract too many terms from data-types associated with messages. Alternatively, data models conceived for being reused may positively impact the precision of registries.
Low cohesive operations in the same port-type	High-level service interface specification	Port-types have weak semantic cohesion.	WSDL documents with low-cohesive port-types convey terms that are not always representative of the domain of their associated services.
Redundant data models	High-level service interface specification	Many data-types for representing the same objects of the problem domain coexist in a WSDL document.	Redundant data models may produce the same effect as redundant port-types. Besides, big and puzzling WSDL documents are exposed to human discoverers.
Ambiguous names	Comments and identifiers	Ambiguous or meaningless names are used for denoting the main elements of a WSDL document.	Syntactic registries gather and preprocess these names, and instead build ambiguous/poor indexes.
Whatever types	Service message exchange	A special data-type is used for representing any object of the problem domain.	Syntactic registries extract few terms (if any) from Whatever types associated with messages.
Empty messages	Service message exchange	Empty messages are used in operations that do not produce outputs nor receive inputs.	Syntactic registries gather and preprocess these names, and instead build ambiguous/poor indexes.

Table 2
Used set of OO metrics.

Metric (acronym)	Meaning	Definition	Source
CBO	Coupling Between Objects	Counts how many methods or instance variables defined by other classes are accessed by a given class.	[9]
WMC	Weighted Methods per Class	Counts the number of methods in a class.	[9]
ATC	Abstract Type Count	Counts the number of method parameters that do not use concrete data-types (i.e., Object), or use Java generics with type variables instantiated with non-concrete data-types.	[24]
EPM	Empty Parameter Methods	Counts the number of methods in a class that do not receive parameters.	[24]

Hypothesis 1 (H_1 : CBO → Enclosed data model). The higher the number of classes directly related to the class implementing a service (CBO (Coupling Between Objects) metric), the more frequent the Enclosed data model anti-pattern occurrences. Code-first tools based on T include in resulting WSDL documents as many XSD definitions as objects are exchanged by service classes methods. Therefore, increasing the number of external objects that are accessed by service classes may increase the likelihood of data-types definitions within WSDL documents.

Hypothesis 2 (H_2 : WMC → Low cohesive operations in the same port-type). The higher the number of public methods belonging to the class implementing a service (WMC (Weighted Methods per Class) metric), the more frequent the Low cohesive operations in the same port-type anti-pattern occurrences. A greater number of methods might increase the probability that any pair of them are unrelated, i.e., having weak cohesion. Since T-based code-first tools map each method to an operation, a higher WMC may increase the possibility that resulting WSDL documents have low cohesive operations.

Hypothesis 3 (H_3 : WMC → Redundant data models). The higher the number of public methods belonging to the class implementing a service (WMC metric), the more frequent the Redundant data models anti-pattern occurrences. Indeed, the number of *message* elements defined within a WSDL document built under T-based code-first tools, is equal to the number of <operation> elements multiplied by two. As each <message> tag may be associated with a data-

type, the likelihood of redundant data-type definitions might increase with the number of public methods, since this in turn increases the number of *operation* elements.

Hypothesis 4 (H_4 : WMC → Ambiguous names). The higher the number of public methods belonging to the class implementing a service (WMC metric), the more frequent the Ambiguous names anti-pattern occurrences. Similarly to H_3, an increment in the number of methods may lift the number of non-representative names within a WSDL document, since for each method a *T*-based code-first tool automatically generates five names (one for the operation, two for input/output messages, and two for data-types).

Hypothesis 5 (H_5 : ATC → Whatever types). The higher the number of method parameters belonging to the class implementing a service that are declared as non-concrete data-types (ATC (Abstract Type Count) metric), the more frequent the Whatever types anti-pattern occurrences. Precisely, the ATC metric was defined after noting that some *T*-based code-first tools map abstract data-types and badly defined generics to xsd:any constructors, being the former the root causes for the Whatever types anti-pattern [27,34].

Hypothesis 6 (H_6 : EPM → Empty messages). The higher the number of public methods belonging to the class implementing a service that do not receive input parameters (EPM (Empty Parameter Methods) metric), the more frequent the Empty messages anti-pattern occurrences. Then, increasing the number of methods without parameters may increase the likelihood of the Empty messages anti-pattern occurrences, because *T*-based code-first tools map this kind of methods onto an operation associated with one input <message> element not conveying XML data.

It is worth noting that a correlation between two variables does not imply causation, i.e., the former variable values are not a sufficient condition for the latter variable ones. However, such a correlation means that the former variable values are a necessary condition for the latter ones. Therefore, WSDL anti-patterns occurrences are not strictly caused by OO metrics values in their associated implementations, but the mentioned WSDL anti-patterns require certain OO metrics values to be present in services implementations.

The Spearman's rank correlation coefficient to model the relations was used. In the tests, a data-set nearly 160 different real code-first services was used, which were collected via the Merobase component finder (http://merobase.com), the Exemplar engine (http://tinyurl.com/7bytzxx), and Google Code (http://code.google.com). Then, OO metrics from service implementations were collected by using an extended version of *ckjm* [37]. For measuring the number of anti-pattern occurrences, the automatic WSDL anti-pattern detection tool mentioned in Section 2.4 was used [32]. Moreover, the hypothesis were tested by separately employing the aforementioned Java to WSDL generation tools, namely EasyWSDL, Java2WS, Java2WSDL and WSProvide. For space reasons, all decisions and validations underpinning this statistical analysis, and the resulting correlation data, can be found in [24,19]. It is worth mentioning that all tools and the experimental data-set are available upon request.

The correlation among the WMC, CBO, ATC and EPM metrics and the six anti-patterns, which was found statistically significant for the analyzed Web Service data-set and the employed generation tools [24], indicates that, in practice, an increment/decrement of the metric values taken on the code of a code-first Web Service directly affects anti-pattern occurrence in its generated WSDL. In this way, some common source code refactorings driven by these metrics can be applied on service implementations as a way to diminish anti-pattern occurrences, namely:

- When considering the WMC metric, a service can be refactored by splitting its operations into two (or more) new services so that on average the metric in the refactored services represents an X% (with X $<$ 100) of the WMC value in the original service. This refactoring is formally known as Extract Class [15].
- When focusing on CBO, the original service implementation can be refactored by replacing some occurrences of a complex data-type for Java primitive types. This refactoring can be seen as the inverse of Replace Data Value with Object [15].
- For the case of the ATC metric, which computes the number of parameters in a class that are declared as Object or data structures – i.e., collections – that do not use Java generics, can also be considered. In the latter case, when this practice is followed, these collections cannot be automatically mapped onto concrete XSD data-types for both the container and the contained data-type in the final WSDL. A similar problem arises with parameters whose data-type is Object. In this sense, a service implementation can be modified in order to reduce ATC by, basically, replacing generic arguments with concrete ones, which is known as Introduce Type Parameter [15].
- Finally, for EPM, which counts the number of methods in a class that do not receive input parameters, the refactoring involves introducing a new parameter (of a primitive data-type) to each of the problematic methods. This refactoring corresponds to Add Parameter [15].

The next subsection illustrates the process of refactoring a sample Web Service by taking into account the above refactoring actions.

Before getting into the practical example, it is worth pointing out that in this paper the term "WSDL anti-pattern" refers to bad specification practices usually found in real-life WSDL documents, as opposed to object-oriented design patterns, which are recurrent but handy (good) design recipes. In terms of their anatomy, while there are some similarities, WSDL anti-patterns are concerned with how the different parts of a WSDL document are specified (e.g., whether ambiguous

```
public interface JTimeLogServer extends java.rmi.Remote {
    public void setUserId(int in0);
    public UserData[] getUserList();
    public TaskData createTask(TaskData in0);
    public List listProject();
    public void deleteProject(Object in0);
    public StatisticData computeStatistics(StatisticData in0);
}
```

Fig. 1. Service code example.

names are used), rather than how the XML elements are combined and structured. Precisely, OO design patterns essentially represent different ways of combining the basic elements of an OO language (i.e., classes and relationships).

As state above, our research shows that some OO metrics taken on service codes have a strong correlation with the manifestation of WSDL-level specification problems. Moreover, Brian Huston [16] has shown that the extent to which software design patterns aimed at an OO design driver (e.g., coupling) are applied to OO codes is congruent with the scores or values computed by OO metrics (e.g., CBO) measuring the same driver. Then, this suggests that there is a correlation between software design patterns and WSDL anti-patterns, and the former could be employed as to avoid the latter. But, our experimental data-set [24], which was built from real service codes, does not evidence a heavy use of OO design patterns so as to ensure probabilistic significance. Consequently, we opted for OO metrics to assess these source codes.

3.1. A practical example

This subsection shows an example applying the refactoring operations to a real service and the impact they have on the generated WSDL documents will be shown. Fig. 1 shows the original version of one of the services from the experimental data-set used during the correlation analysis. It can be seen that the methods in *JTimeLogServer* have dependencies with the types *UserData*, *TaskData* and *StatisticData*. Therefore, the CBO value for the *JTimeLogServer* class is equal to 3. Similarly, the interface defines 6 public methods, thus WMC is equal to 6 in this case. On the other hand, it can be seen that the *listProject* method returns a *List* data-type without using generics to declare the type of objects that the list will contain. This means that the list can hold any type of object. Similarly, the *deleteProject* method receives a parameter of type *Object*, which means that it not possible to know the parameter's concrete type at compile-time. Then, for the *JTimeLogServer* service, the value of the ATC metric is equal to 2. Finally, the EPM metric takes a value of 2 since the methods *getUserList* and *listProject* do not receive any input parameters.

The relevant parts of the WSDL document generated as a result of feeding Java2WSDL with the service of Fig. 1 are shown in Fig. 2. It can be observed that there are 5 defined data-types, namely *UserData*, *StatisticsData*, *PhaseData*, *TaskData* and *RightData*. Therefore, the number of occurrences of the *Enclosed data model* anti-pattern is equal to 5. It is worth noting that the service interface only contained references to 3 complex data-types, as shown in Fig. 1, and yet the resulting WSDL document contains 5 definitions. This stems from the fact that the CBO works exclusively at the service interface level while the generation process has to take into account all transitive dependencies. Then, while the service interface has dependencies with *UserData*, *TaskData* and *StatisticData*, each of these data-types also have extra dependencies that are mapped by the generation tool into the WSDL document, namely *PhaseData* and *RightData*.

Similarly, there are 2 occurrences of the Whatever types anti-pattern since the *xsd:anyType* construct is used on the *deleteProject* and *listProjectResponse* elements. With respect to the Redundant data models anti-pattern, there are 3 pairs of repeated data-types on the WSDL document: [*computeStatistics*; *computeStatisticsResponse*], [*createTask*; *createTaskResponse*] and [*deleteProject*; *listProjectResponse*]. Regarding the Ambiguous names anti-pattern, it can be noted that there are 8 *xsd:element* definitions that correspond to the input and output parameters of each operation. Looking at the *name* attribute of each of these elements shows that non-representative names are used, such as *return* or *args0*. Therefore, there are 8 occurrences of the anti-pattern in this original WSDL document. Additionally, while the first 5 operations of the service deal with the manipulation of user data, the last method computes statistical information, thus introducing an occurrence of the Low cohesive operations in the same port-type anti-pattern. Finally, there are two *message* elements that do not define any content within them, thus producing 2 occurrences of the Empty messages anti-pattern.

After applying the WMC refactoring operation, the original service is split into two new services, as shown in Fig. 3. Each new service now has a WMC value equal to 3, exactly half the value of the original service. The WSDL documents generated for these two new services are depicted in Fig. 4. Following the same analysis made for the original document, it can be seen that the first description depicted in Fig. 4a has only 2 occurrences of the Enclosed data model anti-pattern, while the second document shown in Fig. 4b has 5 occurrences of the same bad practice. Consequently, while the number of occurrences of the anti-pattern remained unchanged for the second service, the first service was improved by removing 3 occurrences of the bad practice. Similarly, now the first WSDL document has no occurrences of the Whatever types bad practice, while the second service has the original 2 occurrences. The Redundant data models anti-pattern follows the same behavior, with 1 occurrence on the first refactored document and 2 occurrences on the second one. Likewise, the Ambiguous names anti-pattern that had 8 occurrences on the original service now presents 4 occurrences on each refactored service. Regarding the Low cohesive operations in the same port-type anti-pattern, the splitting of the service produced the removal of the only occurrence present on the original description. Finally, whereas the original service had 2 occurrences of the Empty messages bad practice, each new service now presents only 1 occurrence.

```
<wsdl:types>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="UserData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="PhaseData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
    <xs:complexType name="RightData">...</xs:complexType>
    <xs:complexType name="StatisticData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="getUserListRequest"/>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

Fig. 2. Original WSDL document generated from Fig. 1.

```
public interface UserManager extends java.rmi.Remote {
    public void setUserId(int userId);
    public UserData[] getUserList();
    public TaskData createTask(TaskData taskData);
}
```

(a) First WMC refactored service

```
public interface ProjectManager extends java.rmi.Remote {
    public List listProject();
    public void deleteProject(Object projectData);
    public StatisticData computeStatistics(StatisticData statisticData);
}
```

(b) Second WMC refactored service

Fig. 3. Resulting service code after the WMC refactoring.

It is worth mentioning that, while it might seem that the number of anti-patterns has not been reduced but rather distributed among the two services, the output of this refactoring operation is the creation of 2 new independent services, each of better quality than the original. As it will be shown in the next section, this has a considerable positive impact on the chances of both services of being discovered by potential consumers. Moreover, this refactoring reduces the total number of occurrences of all anti-patterns and not just those associated with WMC [24], namely Ambiguous names, Low cohesive operations in the same port-type and Redundant data models. It should also be noted that, in order for this refactoring operation to improve the understandability of the resulting services, proper naming best practices and grouping of cohesive methods should be used by developers. Otherwise, the positive effects of the refactoring could be diminished or even result in a negative impact on understandability. The example depicted in Fig. 3 shows how this process was applied while splitting the original service by renaming the resulting interfaces according to the operations they expose.

One might argue that given that refactoring with respect to WMC has so many benefits then the best choice would be to keep splitting the resulting services until there is only one operation on each of them. However, as it is usually the case with software quality attributes, doing so would have several negative points that would likely make the refactoring counterproductive. This would lead to fine-grained services, leading to the well-known Chatty Services prob-

```
<wsdl:types>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="UserData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="getUserListRequest"/>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
```

(a) First WMC refactored WSDL document

```
<wsdl:types>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="PhaseData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
    <xs:complexType name="RightData">...</xs:complexType>
    <xs:complexType name="StatisticData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

(b) Second WMC refactored WSDL document

Fig. 4. Resulting WSDL documents after the WMC refactoring.

lem,[6] which has a negative effect on service consumer application performance. Second, from a service consumer point of view, it is desirable for all the operations related to a coherent functionality to be in the same service, as this makes understanding and using such service easier. Coincidentally, this is a well known good practice in general API design [8]. Finally, from a provider point of view, developing, testing and publishing a service with N logically related operations is intuitively easier and less time consuming that following the same process for N services with individual operations. Then, all these factors and the trade-offs between them have to be considered while refactoring based on the WMC metric.

On the other hand, Fig. 5 shows the result of applying the CBO refactoring on the original service. In this new service, the 3 original complex data-types were replaced by Java's String type. Consequently, the CBO metric value now has a value of 0. The refactored WSDL document generated from this new service, which is depicted in Fig. 6, does not contain any complex data-types definitions. As a result, the Enclosed data model anti-pattern was completely removed. However, the Redundant data models bad practice has increased its number of occurrences from 3 to 7 as the refactored WSDL document now has

[6] http://www.ibm.com/developerworks/webservices/library/ws-antipatterns/.

```
public interface JTimeLogServer extends java.rmi.Remote {
    public void setUserId(int userId);
    public String[] getUserList();
    public String createTask(String task);
    public List listProject();
    public void deleteProject(Object project);
    public String computeStatistics(String statistics);
}
```

Fig. 5. Resulting service code after the CBO refactoring.

```
<wsdl:types>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="getUserListRequest"/>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

Fig. 6. Resulting WSDL document after the CBO refactoring.

```
public interface JTimeLogServer extends java.rmi.Remote {
    public void setUserId(int userId);
    public UserData[] getUserList();
    public TaskData createTask(TaskData taskData);
    public List<Project> listProject();
    public void deleteProject(Project project);
    public StatisticData computeStatistics(StatisticData statisticData);
}
```

Fig. 7. Resulting service code after the ATC refactoring.

7 pairs of repeated data-types: [*computeStatistics*; *computeStatisticsResponse*], [*computeStatistics*; *createTask*], [*computeStatistics*; *createTaskResponse*], [*computeStatisticsResponse*; *createTask*], [*computeStatisticsResponse*, *createTaskResponse*], [*createTask*, *createTaskResponse*] and [*deleteProject*, *listProjectResponse*]. The rest of the anti-patterns remained unchanged. Therefore, even when the 3 occurrences of the Enclosed data model anti-pattern were removed, 4 new occurrences of the Redundant data model were introduced. That is, the refactoring operation had the net effect of increasing the total number of detected anti-patterns on the refactored WSDL document versus the original.

Alternatively, the original service can be refactored taking into account the ATC metric, which results in the service interface shown in Fig. 7. It can be seen that the Object parameter of the *deleteProject* method was replaced by a custom Project object and the List parameter of the *listProject* method was replaced by a generics-aware List<Project> return type. As a result, all the abstract data-types were replaced by concrete ones, decreasing the value of the ATC metric to 0 while still

```
<wsdl:types>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="UserData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="Project"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="Project"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="Project">...</xs:complexType>
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="PhaseData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
    <xs:complexType name="RightData">...</xs:complexType>
    <xs:complexType name="StatisticData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="getUserListRequest"/>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

Fig. 8. Resulting WSDL document after the ATC refactoring.

```
public interface JTimeLogServer extends java.rmi.Remote {
    public void setUserId(int userId);
    public UserData[] getUserList(boolean empty);
    public TaskData createTask(TaskData taskData);
    public List listProject(boolean empty);
    public void deleteProject(Object project);
    public StatisticData computeStatistics(StatisticData statisticData);
}
```

Fig. 9. Resulting service code after the EPM refactoring.

maintaining the benefits of Java's polymorphic types. These refactoring resulted in the complete removal of the Whatever types anti-pattern, as it can be noted by the absence of the *xsd:anyType* construct in Fig. 8. Furthermore, there are less Redundant data models anti-pattern occurrences since the WSDL document now only has 2 pairs of repeated data-types versus the original 3, namely [*computeStatistics*; *computeStatisticsResponse*] and [*createTask*; *createTaskResponse*]. The number of occurrences of the other anti-patterns was not altered.

The last individual refactoring is the one focused on EPM. The effect of applying this operation on the original service is depicted in Fig. 9. As shown in Fig. 10, all the <message> elements on the generated WSDL document now contain a type definition. Therefore, the Empty messages anti-pattern was completely removed. However, the Redundant data models anti-pattern was increased from 3 to 4 occurrences, as now the two added parameters to the *getUserList* and *listProject* operations generate redundant data-types definitions on the description.

Finally, the application of all the individual refactoring operations at the same time on the original service results in the two services shown in Fig. 11 and the corresponding generated WSDL documents depicted in Fig. 12. It can be seen that both documents present no occurrences of the Enclosed data model, Empty messages or Whatever types anti-patterns.

```
<wsdl:types>
  <xs:element name="getUserList">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:boolean"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="UserData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:boolean"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:anyType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="TaskData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="StatisticData"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="PhaseData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
    <xs:complexType name="RightData">...</xs:complexType>
    <xs:complexType name="StatisticData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>...</wsdl:message>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="getUserListRequest"/>...</wsdl:message>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

Fig. 10. Resulting WSDL document after the EPM refactoring.

```
public interface UserManager extends java.rmi.Remote {
    public void setUserId(int userId);
    public String[] getUserList(boolean empty);
    public String createTask(String taskData);
}
```

(a) First refactored service

```
public interface ProjectManager extends java.rmi.Remote {
    public List<String> listProject(boolean empty);
    public void deleteProject(String projectData);
    public String computeStatistics(String statisticData);
}
```

(b) Second refactored service

Fig. 11. Resulting service code after applying all the individual refactorings.

Similarly to the WMC refactoring, the Low cohesive operations in the same port-type occurrence was removed when the original service was split. Conversely, each new description presents 5 occurrences of the Ambiguous names anti-pattern. Finally, the first WSDL document shown in Fig. 12a has 1 occurrence of the Redundant data models while the one depicted in Fig. 12b has 3 occurrences of the same bad practice.

4. Experimental results

This section describes a series of experiments conducted to measure the implications on discovery of early avoiding anti-patterns in service implementations, i.e., performing the refactorings discussed above to remove these bad practices

```
<wsdl:types>
  <xs:element name="getUserList">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:boolean"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="getUserListResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="setUserId">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:int"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTask">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="createTaskResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="getUserListRequest"/>...</wsdl:message>
<wsdl:message name="getUserListResponse">...</wsdl:message>
<wsdl:message name="setUserIdRequest">...</wsdl:message>
<wsdl:message name="createTaskRequest">...</wsdl:message>
<wsdl:message name="createTaskResponse">...</wsdl:message>
```

(a) First refactored WSDL document

```
<wsdl:types>
  <xs:element name="listProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" type="xs:boolean"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="listProjectResponse">
    <xs:complexType>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="deleteProject">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatistics">
    <xs:complexType>
      <xs:element minOccurs="0" name="args0" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="computeStatisticsResponse">
    <xs:complexType>
      <xs:element minOccurs="0" name="return" nillable="true" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:schema attributeFormDefault="qualified">
    <xs:complexType name="UserData">...</xs:complexType>
    <xs:complexType name="PhaseData">...</xs:complexType>
    <xs:complexType name="TaskData">...</xs:complexType>
    <xs:complexType name="RightData">...</xs:complexType>
    <xs:complexType name="StatisticData">...</xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="listProjectRequest"/>...</wsdl:message>
<wsdl:message name="listProjectResponse">...</wsdl:message>
<wsdl:message name="deleteProjectRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsRequest">...</wsdl:message>
<wsdl:message name="computeStatisticsResponse">...</wsdl:message>
```

(b) Second refactored WSDL document

Fig. 12. Resulting WSDL documents after applying all the individual refactorings.

from their associated WSDL documents. The analysis consisted on assessing whether placing effort on refactoring service implementations actually rewards developers by improving their services chances of being discovered or not. If the different refactoring operations actually have a positive impact on discovery, it would also be desirable to know which one is the most efficient in terms of time and effort needed to apply it versus the achieved improvements on service discovery. Moreover, to provide results that are independent of the tool used to generate WSDL documents, and also applicable to various service registries, two different syntactic registries were used in the experiments.

The rest of the section is structured as follows. Section 4.1 explains the experimental conditions. Section 4.2 report on the results achieved with the WSQBE [12] syntactic registry. Section 4.3 presents the results with Lucene4WSDL [10], a syntactic registry based on a modified version of Lucene [21]. Section 4.4 presents a detailed discussion of these results.

4.1. Experimental design

Methodologically, the evaluation consisted of three steps. In a first step, code-first WSDL documents automatically generated from 160 service implementations in the data-set presented in [24] were grouped into six different categories. Five of these categories consisted of WSDL documents generated after applying the refactoring operations listed in the previous section to service implementations in the data-set. These categories were named according to the refactoring operation they included, namely "RefactoredCBO", "RefactoredEPM", "RefactoredATC", "RefactoredWMC" and "RefactoredAll" corresponding to the result of refactoring the original services with respect to CBO, EPM, ATC, WMC or all of the above together, respectively. It is worth noting that the different refactoring operations are independent of each other, which means that when they are all applied on the original data-set at the same time their order is irrelevant. In other words, the "RefactoredAll" data-set is not affected by the order in which the individual refactorings are applied. Another category, named "Original", had the original versions of the improved WSDL documents. The process of grouping WSDL documents in different categories was carried out for each one of the four generation tools employed, namely EasyWSDL, Java2WS, Java2WSDL and WSProvide, providing a total of 24 groups of WSDL documents.

Then, for each pair of *[original;refactored]* group, two different service registries were supplied with both categories of WSDL documents. Note that each of these groups were generated using the four code-first tools, thus resulting in four different groups for each refactoring operation. The service registries returned an ordered list of WSDL documents relevant to a given query, being the WSDL document at the top of the list the most relevant, and so on. The rationale behind this decision was to study the effects of each refactoring operation with respect to the original version of the same documents, independently from each other.

Finally, the employed registries were queried using one query per available service operation in the published services. For each query, the Precision-at-*n* metric was employed to measure the position on which either the original or the refactored WSDL documents were retrieved. Precision-at-*n* computes precision at different cut-off points of the results list [18]. For example, if the top 5 documents are all relevant to a query and the next 5 are all non-relevant, we have a precision of 100% at a cut-off of 5 documents but a precision of 50% at a cut-off of 10 documents. Formally:

$$Precision\ at\ n = \frac{RetRel_n}{n}$$

where $RetRel_n$ is the total number of relevant services retrieved in the top n positions. It was decided to employ this metric as the main metric because it does not only characterize how well a search engine performs in finding relevant documents, or services in this case, but it also takes into account the position of each relevant retrieved service within the result list. This fact makes this metric especially suitable for comparing registries that retrieve services in a rank, like WSDL and Lucene4WSDL do.

An important issue when developing a service-oriented application is deciding whether to implement some application component, or reuse an existing implementation instead. In the latter case it is said that the service is *outsourced*. However, in order to outsource a service it must first be discovered. In this sense, it can be assumed that if developers want to replace an operation with a functional equivalent operation provided by an external service, they will probably use the name of the replaceable operation as a query. This assumption is analogous to the Query-By-Example concept [11,12], upon which WSQBE is built. For example, the query for looking for operations whose signature is "getActiveWorkflows(userID:string)" may be "get active workflows". In fact, WSQBE splits combined words within queries. Following this assumption, 879 queries were built from the source code of original service implementations, one per offered operation. Then, two WSDL documents were associated with each query, one document belonging to the original service category and another from the refactored one. For the association, the WSDL documents containing the operation needed were statically selected.

At this point it should be explained why discovering a refactored WSDL document could potentially differ, in terms of retrievability, from discovering its original WSDL counterpart upon the same query (Web Service operation names in this case), which motivates our experiments. Apart from affecting anti-pattern occurrences in generated WSDL documents (see Section 3.1), the refactorings explained so far also affect documents from a pure IR perspective. Any syntactic registry operates by extracting representative words from WSDL documents and then associating these words together with newly published documents into a unique index. The registry also computes the relative weight or importance a word has within a document. Then, upon a user's query (list of words), the syntactic registry searches the index taking into account the document-word(s) associations in the index and the weights.

As a consequence, this index is very sensitive to changes in the content of WSDL documents and hence two similar documents might not be associated to the exact set of words (or weights) within the same index. As a corollary, the same syntactic query can produce different retrieval results. Precisely, our refactorings literally lead to changes in the words of generated WSDL documents. WMC causes a WSDL document to be split into two or more documents, which are treated as different indexing units by the registry. EPM causes the document to have more terms per "void" operation, whereas CBO reduces the number of words since complex data-types (named after user classes) are replaced by a smaller, fixed set of terms (i.e., those associated to primitive types). Finally, the ATC refactoring causes the opposite of what CBO does, since replacing "Object" or instantiating Java templates with concrete user classes leads to WSDL documents with more terms, since classes are translated to XSD data-types having representative names, potentially with nested structures.

Returning to the retrievability metrics employed, the Precision-at-n results were computed for each query with n in [1–10], and a fixed result window size of 10. This window size was chosen because a good balance between the number of candidates and the number of relevant candidates retrieved is desirable. Moreover, it is assumed that a developer can easily examine 10 Web Service descriptions in a usual discovery scenario. Therefore, by setting the window size to 10, the considered number of relevant services in the result list is up to 10 candidates. After publishing each *[original;refactored]* group for each code-first generation tool, this metric was computed for every query and finally averaged over the 879 queries built.

Besides Precision-at-n, the Recall-at-n and Discounted Cumulative Gain (DCG) measures have been calculated, again with a fixed window size 10. Recall-at-n computes how effective a service registry is in retrieving relevant documents in a window of n documents [18]. Formally:

$$Recall\ at\ n = \frac{RetRel_n}{R}$$

where R is the total number of relevant services to a query in the collection. By returning every document in the collection for every query a recall of 100% could be achieved, but looking for relevant services in the entire collection would be a cumbersome task. Conversely, as previously described for the values of Precision-at-n, only up to 10 candidates are considered for every query, i.e., $RetRel_n = RetRel_{10}$. For the sake of brevity, in the rest of this section the Recall-at-n metric will be referred to as Recall.

On the other hand, DCG is a measure for ranking quality and measures the usefulness or gain of an item based on its relevance and position in the provided list of candidates. A higher DCG value means that a query returned a list of better ranked candidates. Formally, DCG is defined as:

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

where p is the size of the candidate list, which for these experiments is 10, and rel_i indicates if the candidate retrieved in the ith position of the list was relevant. The DCG values for all queries can be averaged to obtain a measure of the average performance of a ranking algorithm, named Normalized DCG (nDCG). Then, in the experiments the DCG metric was calculated for each query and then averaged over the 879 built queries to produce the nDCG values.

In principle, only original and refactored versions of the same WSDL documents coexist in a registry, which is useful for comparison purposes but it is somewhat unrealistic. Thus, a second evaluation was made to assess the implications of applying the proposed refactorings in a more realistic scenario, by reproducing the same experiment with the data-set of 1664 publicly available WSDL documents described in [3]. This data-set was published on the two employed registries along with each pair of *[original;refactored]* data-set of WSDL documents. Therefore, this experiment was equal to the former except for the second step, which has been modified to simulate a real world scenario. The following sections will describe the results obtained with each registry.

4.2. WSQBE

The first employed registry was WSQBE, an academic approach to Web Service retrieval that combines classic IR techniques with a search-space reduction mechanism. As described earlier, the idea is to represent Web service descriptions and queries within a classic vector space [35], which is partitioned into sub-spaces corresponding to the categories of the services. Then, instead of performing a *one-to-many* matching (i.e., matching the query vector against all vectors in the search space), WSQBE proposes a two-step matching approach that first reduces the search space to a sub-space and then compares a query against service descriptions belonging to this small sub-space. This approach makes WSQBE appropriate for dealing with a large number of services. Moreover, WSQBE allows human discoverers to ask for a third-party service by providing a handful set of keywords of its expected functionality.

On a first set of experiments, the WSQBE registry was supplied with each pair of *[original;refactored]* data-set for each of the generation tools under study. In other words, no "noise" was introduced as only the original and refactored versions of the same data-set were present in the registry. In this context the term noise refers to the presence of WSDL documents that are not part of the experimental data-set but are published in the registry to establish a practical scenario. It should be noted that the registry was wiped after each experiment, which means that the different *[original;refactored]* groups for different refactoring operations and different generation tools never coexisted in the registry. In this way, the effects of the code-first refactorings are analyzed independently for each individual refactoring and tool with respect to the original data-set. Table 3 shows the Precision-at-1, Recall and nDCG metrics respectively for each tool and data-set, namely *[original;RefactoredAll]* (DS_1), *[original;RefactoredCBO]* (DS_2), *[original;RefactoredEPM]* (DS_3), *[original;RefactoredATC]* (DS_4) and *[original;RefactoredWMC]* (DS_5). It is worth noting that, although Precision-at-n for n in [1–10] was computed, data tables from now own will specifically depict Precision-at-1 and figures will depict Precision-at-n for n in [1–10]. The first cut-off point – i.e., n equals to 1 – is extremely important because having a higher Precision-at-1 means that a registry performs better in retrieving a relevant service at the top of the result list. As supported by different experiments, higher values for Precision-at-1 have a great impact on retrievability because users tend to select higher ranked search results first [1]. For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the second ranked one is, at most, 60% [1].

Table 3
WSQBE **without** noise data-set Precision-at-1 (top subtable), recall (center subtable) and nDCG (bottom subtable).

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[16.0;53.7]%	[10.0;64.3]%	[35.3;39.0]%	[7.0;67.3]%	[15.9;53.7]%
Java2WS	[16.3;53.0]%	[20.4;53.8]%	[40.1;34.1]%	[9.0;64.9]%	[16.0;53.3]%
Java2WSDL	[26.6;43.2]%	[6.5;67.6]%	[33.9;40.2]%	[7.3;66.8]%	[25.7;44.3]%
WSProvide	[15.8;53.3]%	[19.6;54.6]%	[36.1;38.1]%	[5.5;68.7]%	[15.8;53.3]%

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[92.4;91.4]%	[92.6;92.6]%	[92.5;92.5]%	[92.5;92.5]%	[92.1;91.8]%
Java2WS	[92.0;91.5]%	[92.5;92.5]%	[92.5;92.5]%	[92.2;92.2]%	[92.1;91.7]%
Java2WSDL	[91.3;90.1]%	[92.2;92.2]%	[92.2;92.2]%	[92.2;92.2]%	[92.0;91.9]%
WSProvide	[90.8;88.9]%	[92.5;92.5]%	[92.5;92.5]%	[92.5;92.5]%	[92.1;91.5]%

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[80.1;82.3]%	[82.8;84.6]%	[83.9;83.5]%	[82.8;84.4]%	[80.0;82.6]%
Java2WS	[80.1;82.2]%	[82.9;84.1]%	[83.5;83.4]%	[82.8;83.8]%	[80.1;82.5]%
Java2WSDL	[80.5;78.8]%	[82.9;83.8]%	[83.9;82.8]%	[82.7;84.0]%	[81.0;81.1]%
WSProvide	[79.4;80.3]%	[83.3;84.0]%	[83.4;84.0]%	[83.1;84.2]%	[80.1;82.3]%

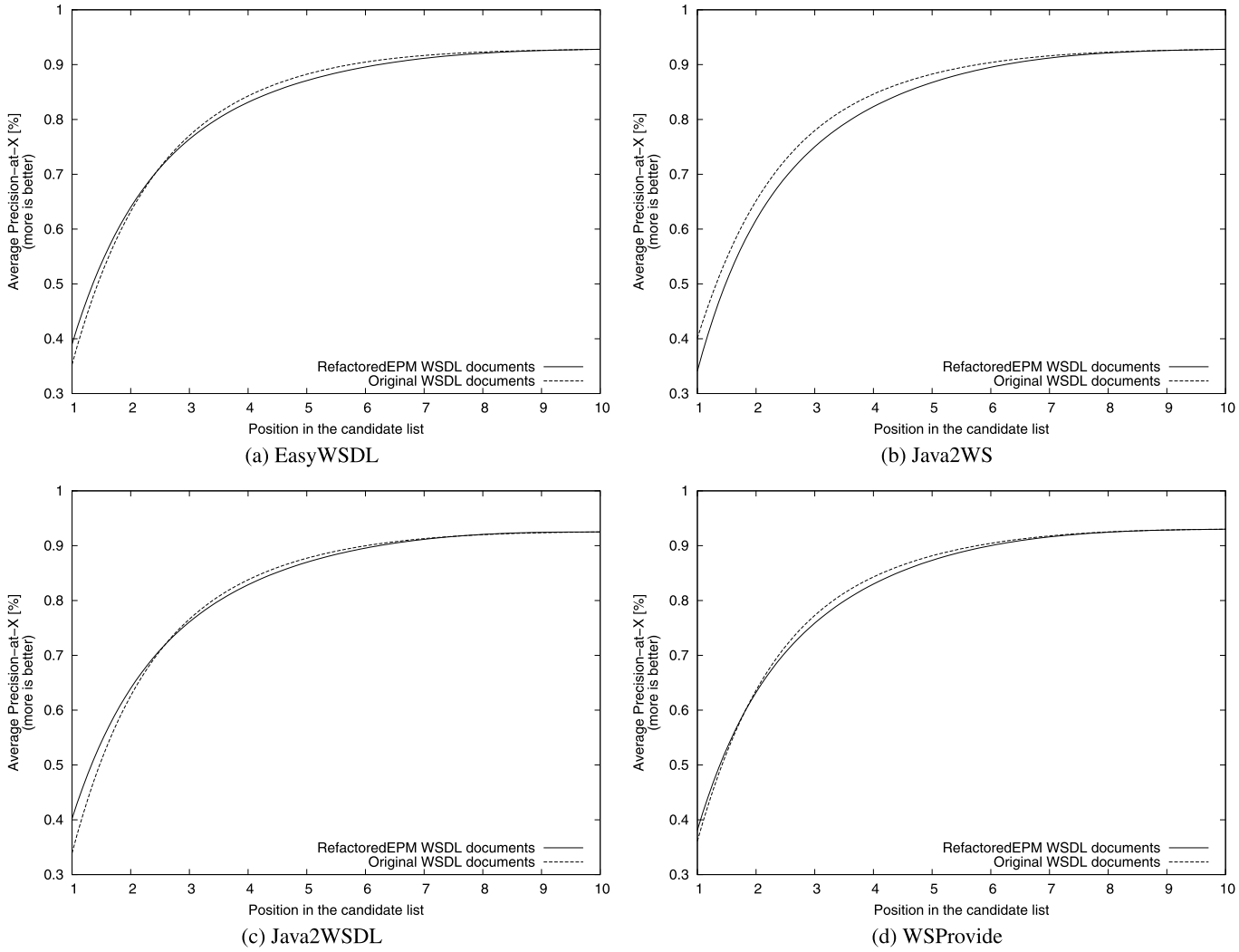


Fig. 13. EPM impact on retrievability when using WSQBE **without** noise.

Several interesting facts arose from these results. First, it can be observed from Table 3 (top subtable) that even when the EPM refactored documents presented slightly better results on average when compared to the original ones, the difference between the two is not highly significant. Concretely, the Precision-at-1 values were increased by 3.7%, 6.3% and 2% for EasyWSDL, Java2WSDL and WSProvide, respectively. Moreover, when Java2WS was used, the original documents showed a higher Precision-at-1 value than their refactored versions. Fig. 13 shows the Precision-at-x metric for x

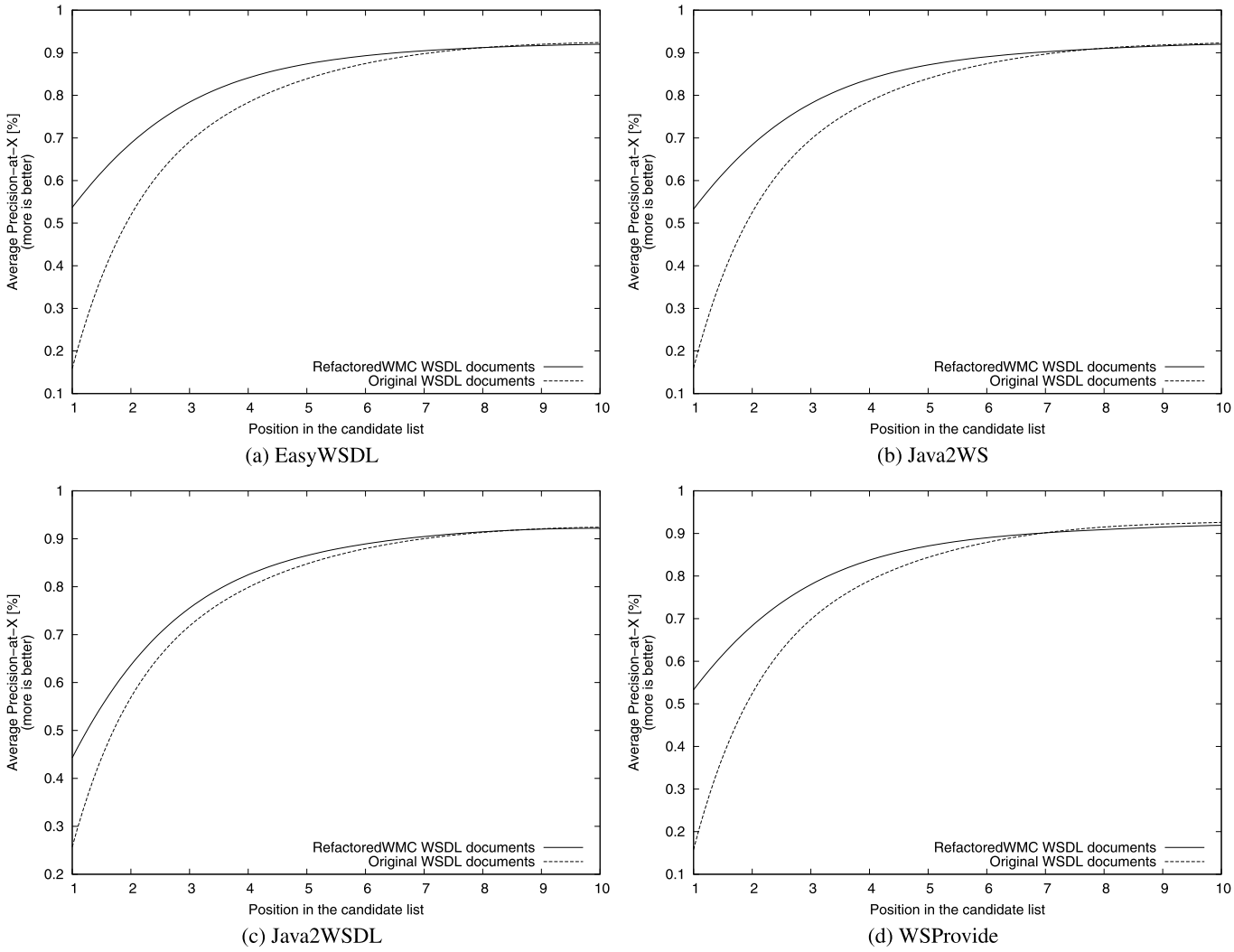


Fig. 14. WMC impact on retrievability when using WSQBE **without** noise.

in $[1, 10]$ for each generation tool when EPM is considered. These results suggest that putting the focus on EPM while applying the refactoring operations might not be a very efficient choice from the point of view of service retrievability, since the increase on retrieval effectiveness is minimal. At the same time, with respect to the other metrics, the refactored services were competitive in terms of retrievability, and no significant differences were obtained. Below we focus on Precision-at-n.

On the other hand, Table 3 shows that when the WMC refactoring is considered there is a significant improvement on retrievability with respect to the original data-set, as shown by the increase on the Precision-at-1 values. Furthermore, this result is consistent across the four different generation tools with increases on the Precision-at-1 values of 37.8%, 37.3%, 18.6% and 37.5% for EasyWSDL, Java2WS, Java2WSDL and WSProvide respectively. Fig. 14 shows this improvement on service retrievability.

Similar results were obtained for the case of CBO and ATC, as shown in Figs. 16 and 15, respectively. Furthermore, for these two metrics the improvement on service retrievability is greater than the one obtained when WMC was considered. The Precision-at-1 values for these two refactorings from Table 3 suggest that, from a retrievability perspective, focusing on the ATC metric for the refactoring process results in the most efficient choice when only the original and refactored versions of the same data-set are published on the registry, followed closely by CBO.

Finally, while combining all the refactorings on the original data-set has a positive impact on service retrievability, as depicted in Fig. 17, this improvement is smaller that the one obtained by using the individual refactorings previously shown. These results are consistent with previous results [24], where the application of individual refactoring operations had a greater impact on the number of anti-patterns when compared to the combination of all the refactorings applied on the same data-set. For example, when Java2WSDL was used and the original services were refactored according to the WMC metric, the average number of anti-patterns was reduced from 102.66 to 37.58 occurrences. However, when all the refactorings were applied on the original services the average number of anti-pattern occurrences was 51.59.

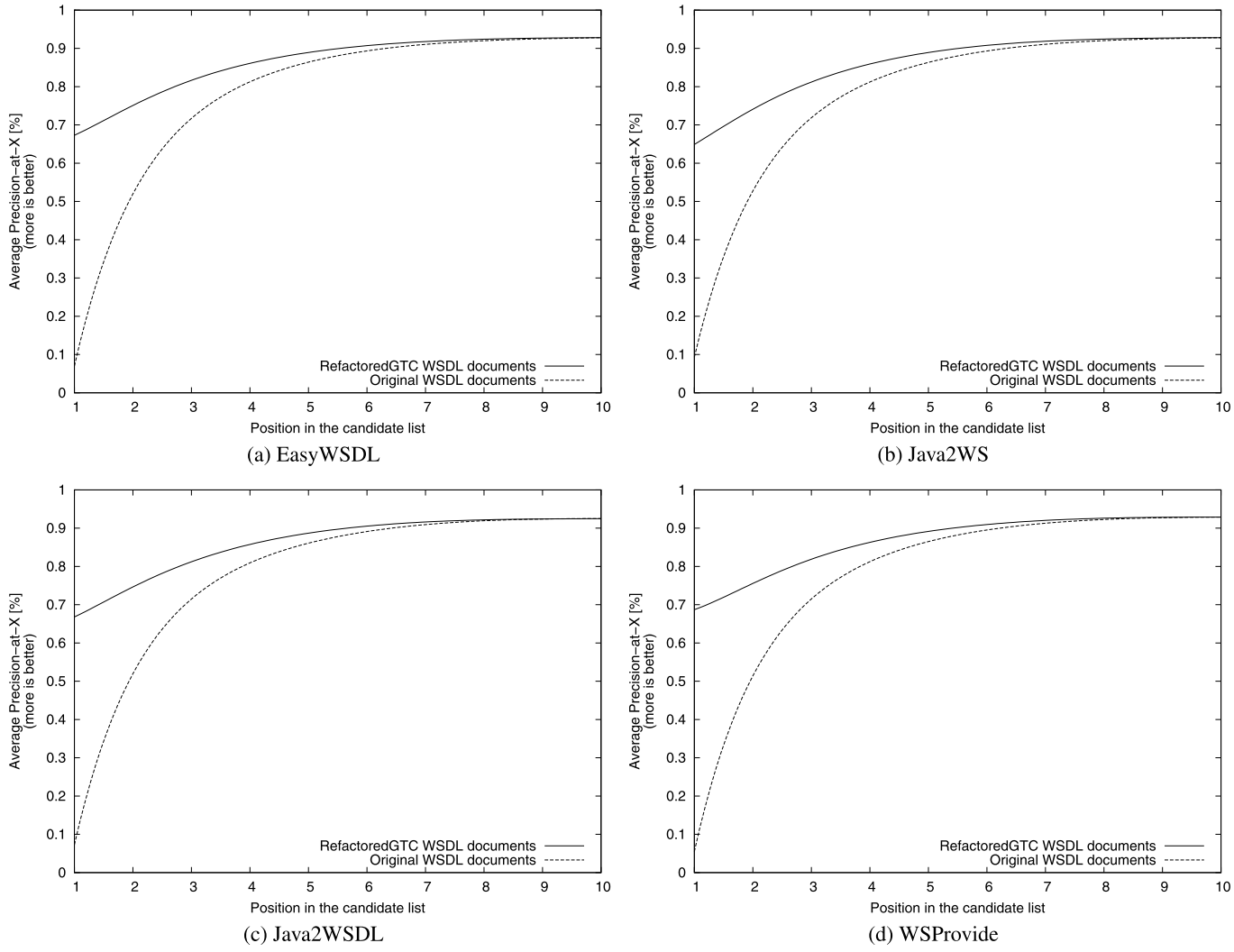


Fig. 15. ATC impact on retrievability when using WSQBE **without** noise.

Note that, according to Table 3, the refactorings had little effect on Recall and nDCG. This is sound since these two metrics are window size-sensitive. This is, the search window size was set to 10, and the maximum number of relevant WSDL documents per query is always equals to 2 (one original and one refactored WSDL document). Thus, there is a high probability that both type of documents fit in the results list. Despite not being useful for spotting performance differences, the aim of using these two IR metrics was to show that the refactorings do not compromise the relevance of the WSDL documents retrieved, and at the same time the refactorings offer gains in terms of positional precision.

Despite these promising results, as it was mentioned in Section 4.1, the fact that only two versions of the same data-set are deployed on a registry is not realistic. For this reason, the same set of experiments was repeated but in addition to each original and refactored data-set, a third data-set of 1664 WSDL documents was published in the registry. The values of the Precision-at-1, Recall and nDCG metrics are shown in Table 4.

There are some remarkable similarities between Table 4 and Table 3 when comparing the Precision-at-1 subtables. When EPM is considered, the improvement obtained is negligible for the case of WSProvide, and for the remaining three tools it produces worse Precision-at-1 values than the original data-set, thus making EPM a poor choice when deciding which refactoring operation to apply. This situation is depicted in Fig. 18. Likewise, the use of WMC to drive the code-first refactorings results on a significant increase on the precision of the employed registry, as shown in Fig. 19. Furthermore, the increase on the Precision-at-1 metric value for each WSDL generation tool is similar to the one obtained in the previous set of experiments. Concretely, when only the original and refactored data-sets were published on the registry, Precision-at-1 was increased by 37.8%, 37.3%, 18.6% and 37.5% for EasyWSDL, Java2WS, Java2WSDL and WSProvide respectively. On the other hand, when the noise data-set was published together with the original and refactored ones, the same metric was increased by 35.5%, 32.8%, 11.1%, and 32.8% for the same tools mentioned previously. These results suggest that WMC is an excellent choice to drive the code-first code refactorings, since the change on the services code is fairly simple and the positive effect on service retrieval is significant in a real world scenario and consistent across different WSDL generation tools.

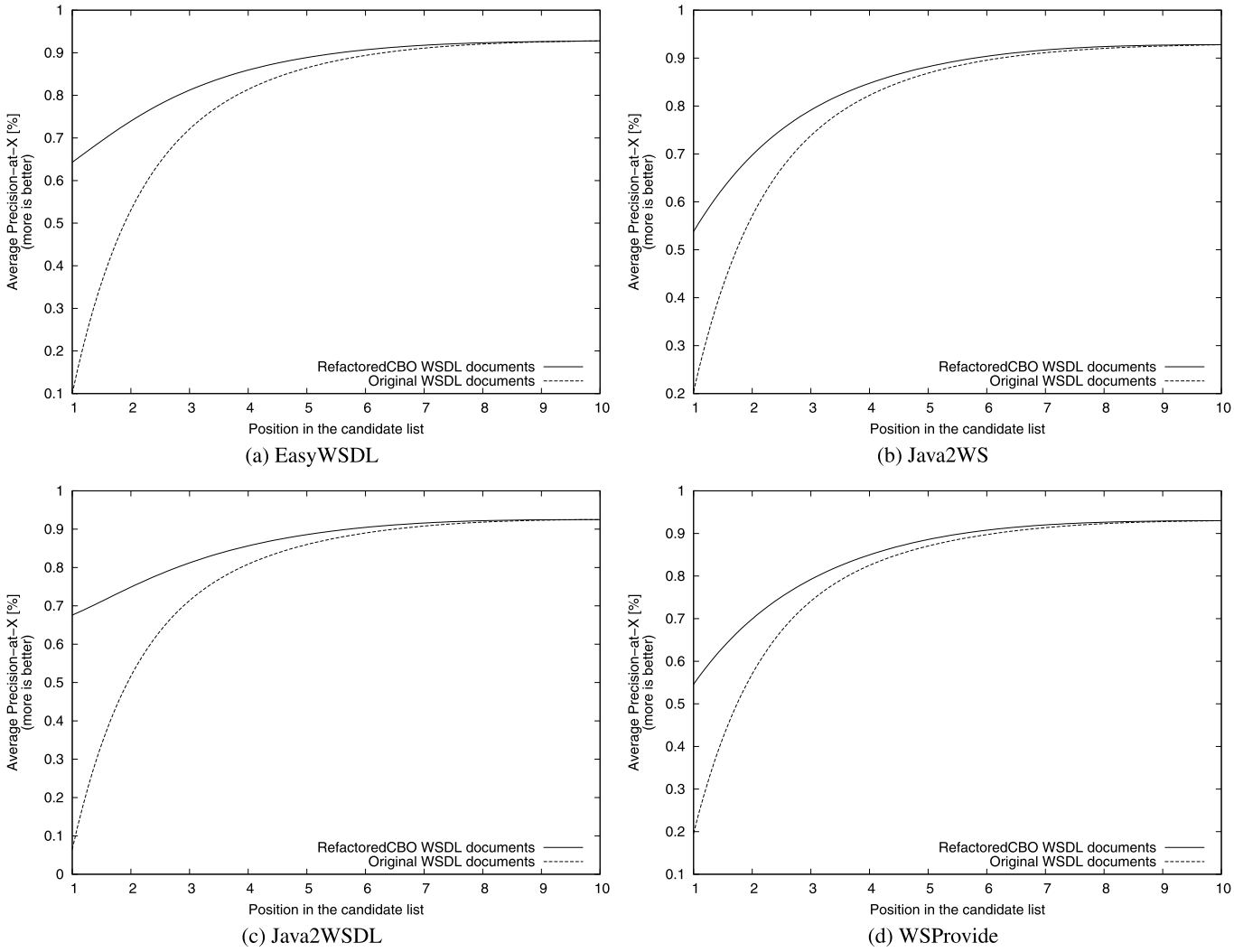


Fig. 16. CBO impact on retrievability when using WSQBE **without** noise.

Unlike the results obtained when only the original and refactored data-sets were published on the registry, Precision-at-1 in Table 4 shows that in a more real scenario the CBO and ATC refactorings do not produce a highly significant improvement in retrievability, as shown in Figs. 21 and 20. These results are further discussed in Section 4.4 but clearly the outcome obtained when an external data-set was published in the registry along with the original and refactored ones is more valuable from a practical perspective, as developers' services published in public registries usually have to coexist with other services that are not under their control.

Finally, it is worth noting that the results obtained when all the refactorings were applied on the same data-set are analogous to those showed in Table 3. This is, there is an improvement but the positive effect of some of the individual refactorings outweighs the effect of applying all the refactorings together. Fig. 22 depicts this situation.

4.3. Lucene4WSDL

In order to ensure that the results shown on the previous section were not dependent on the employed registry, a second experiment was conducted using Lucene4WSDL. As it was explained in Section 4.1, this new registry is a slight adaptation of the well known text search engine Lucene. Lucene is an open-source software that follows a classic IR-based approach to perform full-text search on text documents.

Following the same methodology used for WSQBE described in Section 4.2, two different sets of experiment were carried out. The first set measures the impact of each refactoring when only the original and corresponding refactored data-set were published on the registry, while the second set of experiments does the same when the additional data-set of 1664 WSDL documents described in the previous section is also published. Table 5 shows the Precision-at-1, Recall and nDCG metrics for the former set of experiments.

Similarly, Table 6 depicts the same metrics for the latter set of experiments, i.e., with noise. Interestingly, the results obtained in both sets of experiments are consistent with those obtained with WSQBE. Concretely, the top subtables in Tables 5 and 6 show that in both cases the EPM refactoring resulted in an average increase on Precision-at-1 of 4% when

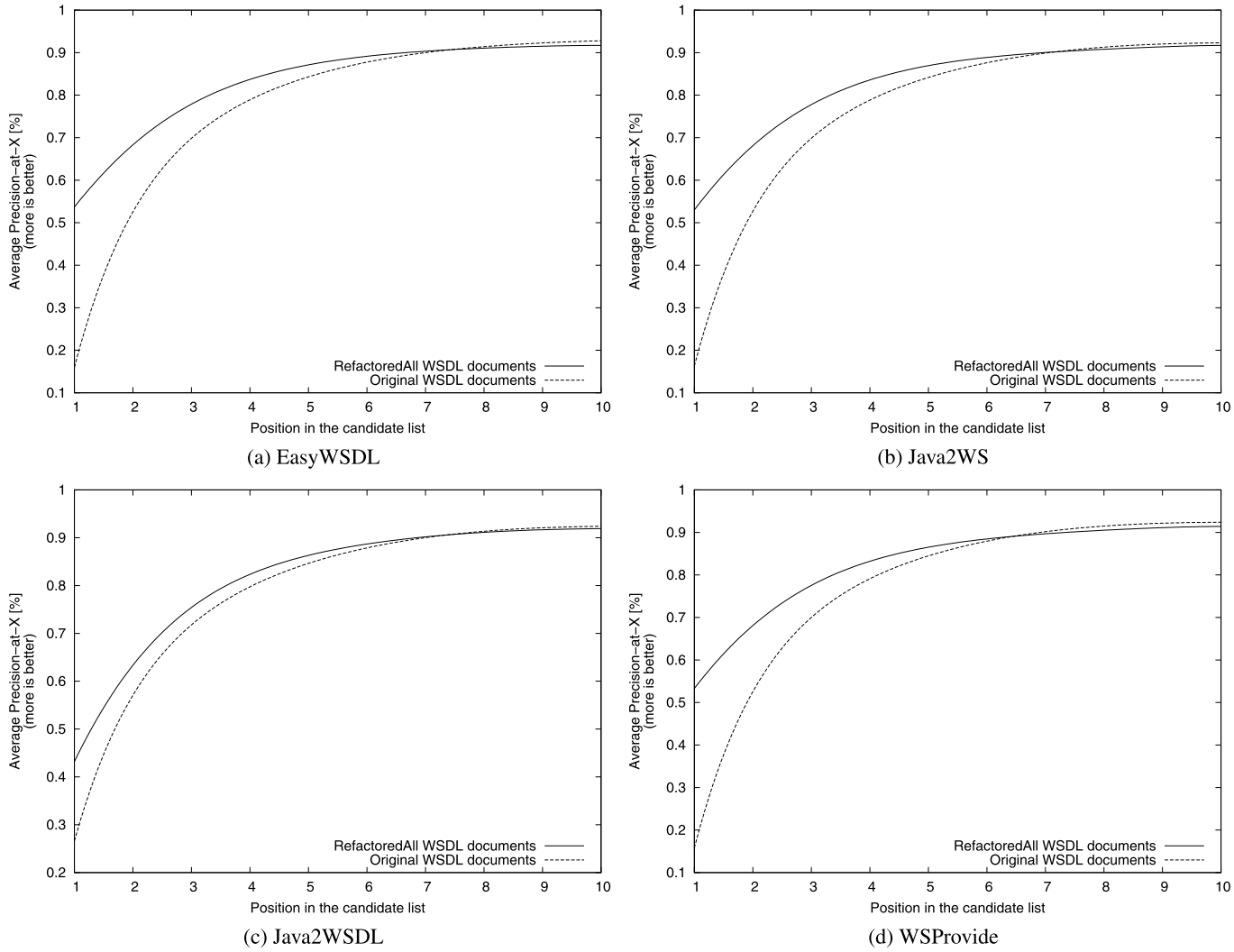
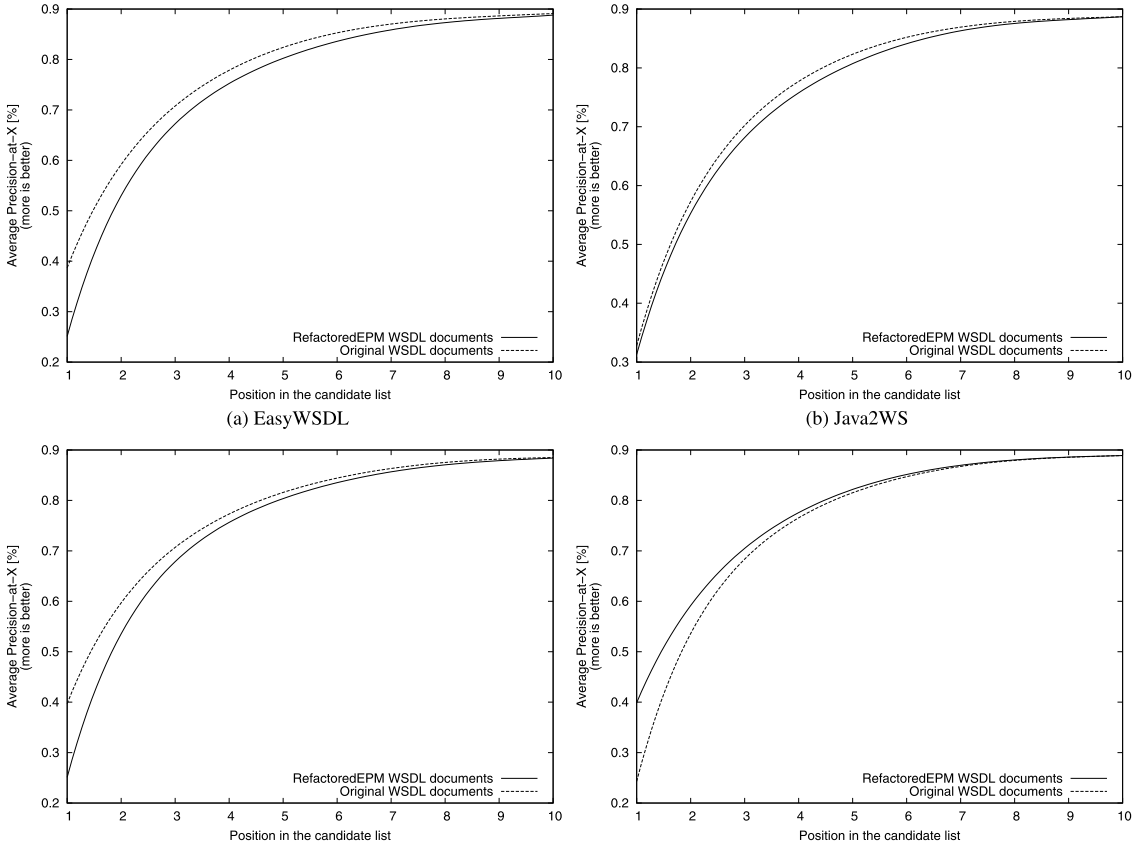


Fig. 17. Impact on retrievability when combining all refactorings and using WSQBE **without** noise.

Table 4
WSQBE **with** noise data-set Precision-at-1 (top subtable), recall (center subtable), and nDCG (bottom subtable).

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[14.8;48.2]%	[22.5;41.2]%	[38.7;25.1]%	[24.1;39.6]%	[13.7;49.2]%
Java2WS	[16.1;47.6]%	[20.8;43.4]%	[32.8;31.3]%	[24.0;40.2]%	[15.3;48.1]%
Java2WSDL	[27.3;36.6]%	[38.6;26.4]%	[39.7;25.1]%	[30.0;34.9]%	[26.2;37.3]%
WSProvide	[15.8;47.5]%	[29.2;35.2]%	[24.3;39.9]%	[23.2;41.1]%	[15.2;48.0]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[87.8;88.6]%	[88.6;88.4]%	[89.0;88.4]%	[88.2;88.5]%	[87.9;88.9]%
Java2WS	[87.9;88.3]%	[88.3;88.5]%	[88.5;88.3]%	[88.4;88.4]%	[88.1;88.6]%
Java2WSDL	[88.1;87.4]%	[88.1;88.1]%	[88.3;88.0]%	[88.1;88.1]%	[87.8;87.5]%
WSProvide	[88.1;88.2]%	[88.5;88.3]%	[88.2;88.4]%	[88.6;88.2]%	[88.1;88.2]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[73.4;77.0]%	[76.6;76.7]%	[77.6;76.2]%	[76.5;77.0]%	[73.3;77.1]%
Java2WS	[74.1;76.5]%	[76.3;77.5]%	[77.9;76.0]%	[76.4;77.5]%	[74.0;77.0]%
Java2WSDL	[74.7;75.9]%	[76.2;77.6]%	[78.3;75.7]%	[76.8;77.0]%	[74.1;76.0]%
WSProvide	[74.1;76.6]%	[76.8;77.4]%	[76.6;77.5]%	[76.8;77.4]%	[73.9;76.7]%

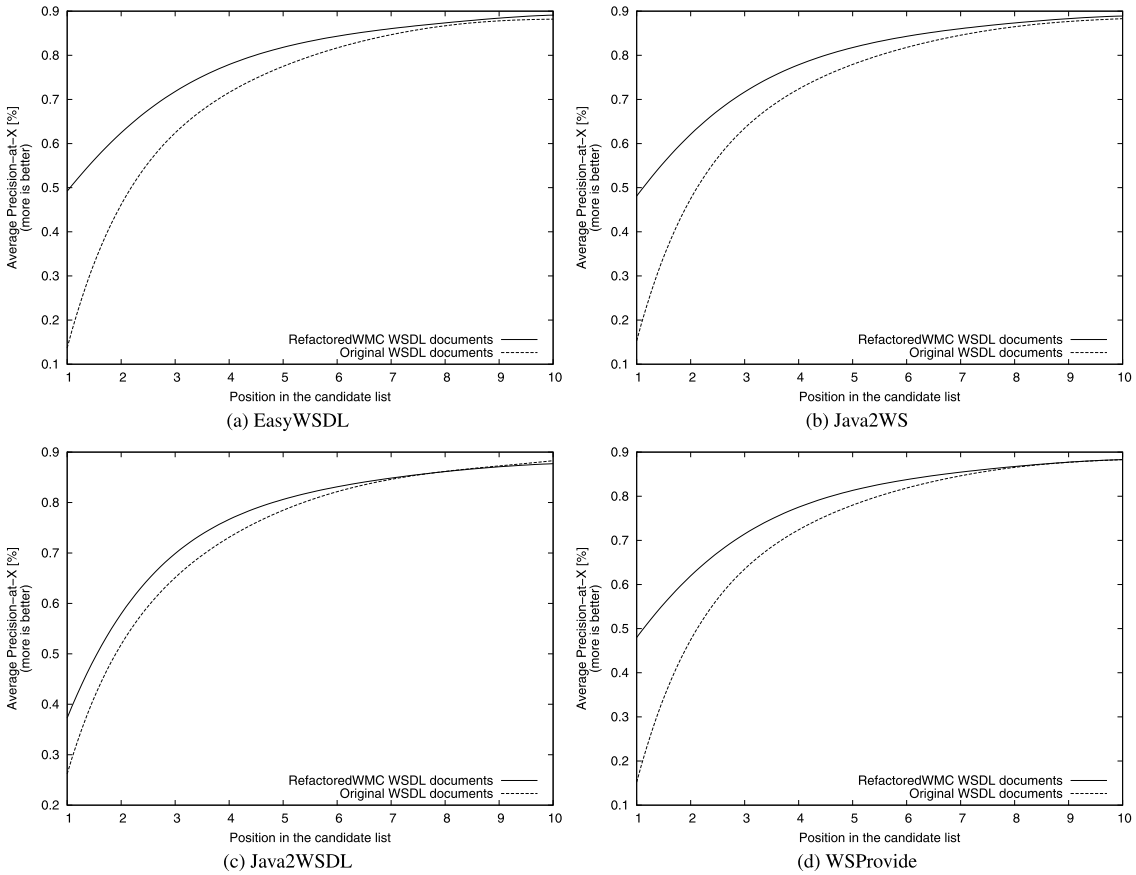
only the original and refactored data-set were published on the registry and an average increase of 5% when the noise data-set was considered. This is depicted in Figs. 23 and 27. It is worth noting that these results are analogous to the ones obtained with WSQBE, where the EPM refactoring was shown to have a low impact on service retrievability. Similar results were obtained for ATC and CBO, as shown in Figs. 25 and 26 for the first set of experiments and Figs. 29 and 30 for the



(a) EasyWSDL (b) Java2WS

(c) Java2WSDL (d) WSProvide

Fig. 18. EPM impact on retrievability when using WSQBE **with** noise.



(a) EasyWSDL (b) Java2WS

(c) Java2WSDL (d) WSProvide

Fig. 19. WMC impact on retrievability when using WSQBE **with** noise.

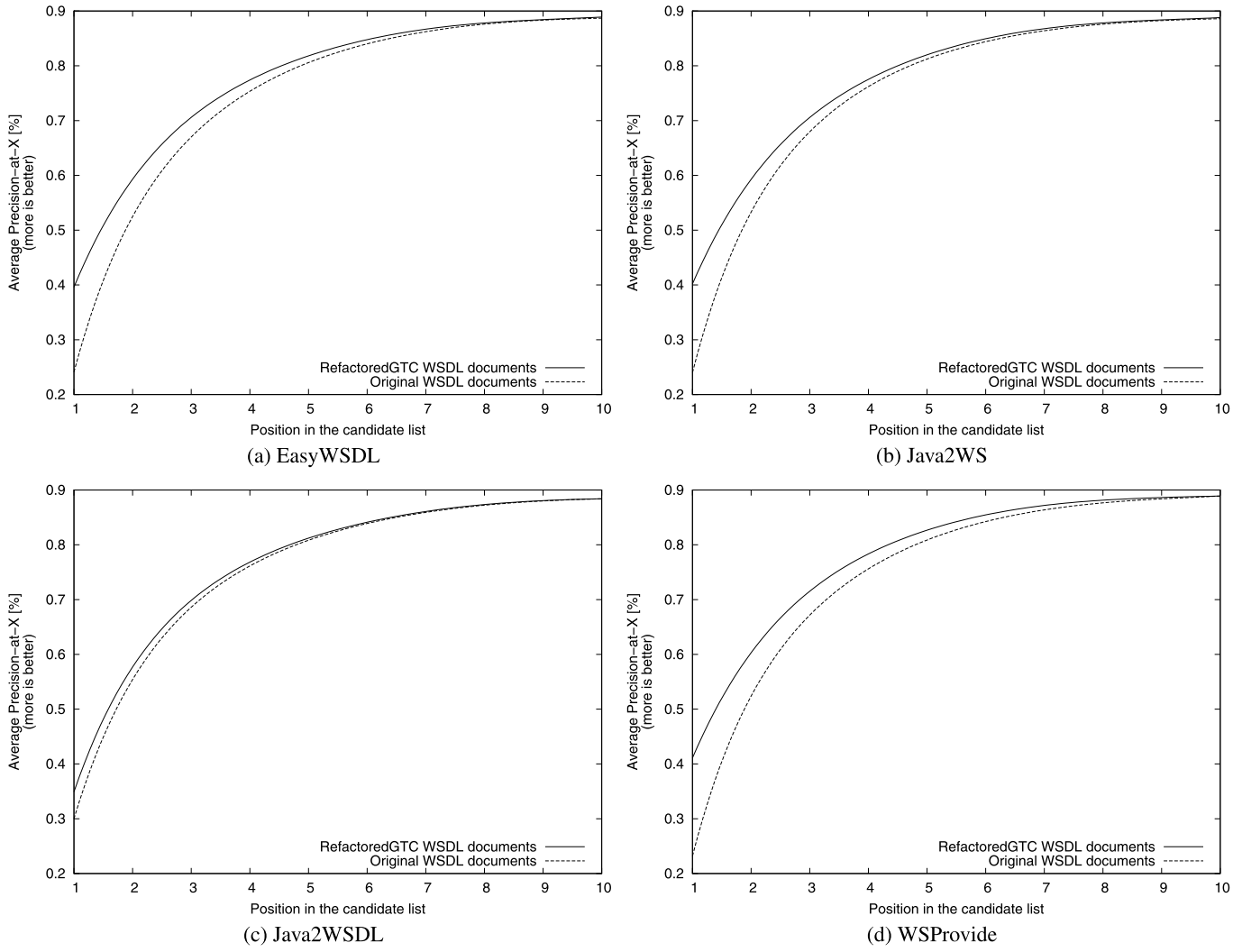


Fig. 20. ATC impact on retrievability when using WSQBE **with** noise.

second set. In addition, there are not significant differences in terms of the Recall and the nDCG metrics, although refactored services tend to have higher nDCG for Lucene4WS compared to WSQBE. Therefore, we again analyze Precision-at-*n*. For space reasons, and since "RefactoredAll" did not introduce significant improvements compared to individual refactorings, we omit the associated graphics.

On the other hand, the top subtable in Table 5 shows that when the services are refactored following WMC, a consistent and significant improvement can be observed on their efficiency for all the employed generation tools, as depicted in Fig. 24. Likewise, when the noise data-set is added to the registry, the same significant improvement can be observed on the Precision-at-1 values shown in the top subtable in Table 6 and depicted in Fig. 28. Finally, for both groups of experiments, when all the refactorings were combined the improvement on retrievability was not as good as the one obtained for WMC. Again, these result are analogous to the ones obtained for the same metrics with WSQBE.

4.4. Discussion

The previous sections showed that by refactoring services implementation code according to the studied refactoring operations the retrieval effectiveness of the two syntactic registries was improved. Furthermore, it was shown that these results were consistent when all the different generation tools were employed. This fact evidences that the improvement on service retrieval was not caused by the underlying mechanism used by the different registries or code-first tools employed but rather by the removal of anti-patterns caused by the use of the refactoring operations.

As argued in [34], these positive results may stem from the fact that the original WSDL documents usually contain redundant and non-specific terms, i.e., when there is no a strong connection between the operation functionality and the terms used to define it. Certainly, the removal of anti-patterns occurrences results in WSDL documents that are more concise and specific. Therefore, it is expected for a specific relevant service to be retrieved near to the top of the result list when

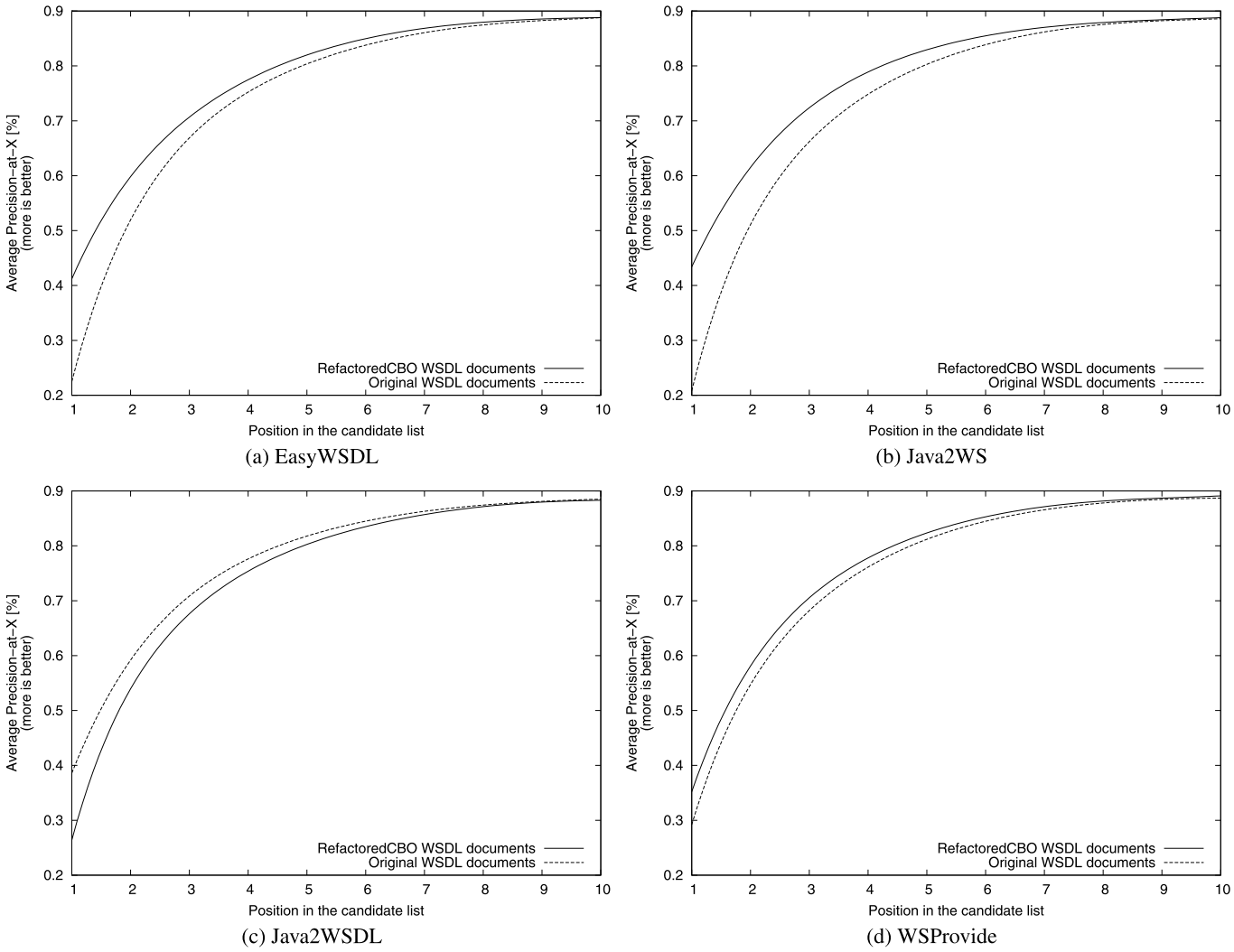


Fig. 21. CBO impact on retrievability when using WSQBE **with** noise.

specific and representative service descriptions have been published in the registry, which is the case of the improved WSDL documents.

However, the goal of the approach to removing WSDL anti-patterns from code-first Web Services in this paper is two-fold. On one hand it tries to obtain more clear, concise and unambiguous descriptions that are easier to understand by developers who are trying to determine what operations from a Web Service they need to use and how they should invoke it. The second goal is exclusively aimed to help syntactic registries to better process the resulting WSDL documents and improve their retrieval efficiency. Keeping these two goals in mind, this section will discuss situations where a refactoring operation may not be ideal from a retrieval perspective but their use would still be beneficial from the point of view of a consumer's ability to understand the resulting descriptions. In those cases the decision of whether the refactoring should be applied or not, like most situations in software involving different quality attributes, involves a trade-off between the two mentioned goals.

Particularly, the EPM refactoring did not have a significant impact on service retrieval. Moreover, in some cases the refactoring operation resulted in the worst retrieval efficiency. The fact that this result is consistent across the different syntactic registries and code-first generation tools analyzed suggests that, this refactoring operation should not be applied provided that the goal is to improve code-first service retrieval. To better understand this behavior, it is worth remembering that the EPM refactoring consisted in introducing an additional, potentially unrelated parameter to those methods on the services implementation code that did not receive any parameters. Consequently, when the syntactic registries preprocess the refactored WSDL documents, the new parameter adds unnecessary noise that deteriorates the efficiency of the registries when answering queries. This is also consistent with the effect of the EPM refactoring on the total number of anti-patterns. This refactoring in general produces a small reduction on the total number of occurrences of anti-patterns on a data-set [24]. Therefore, it is to be expected that applying the EPM refactoring to our experimental data-set has little or even negative impact on service retrieval.

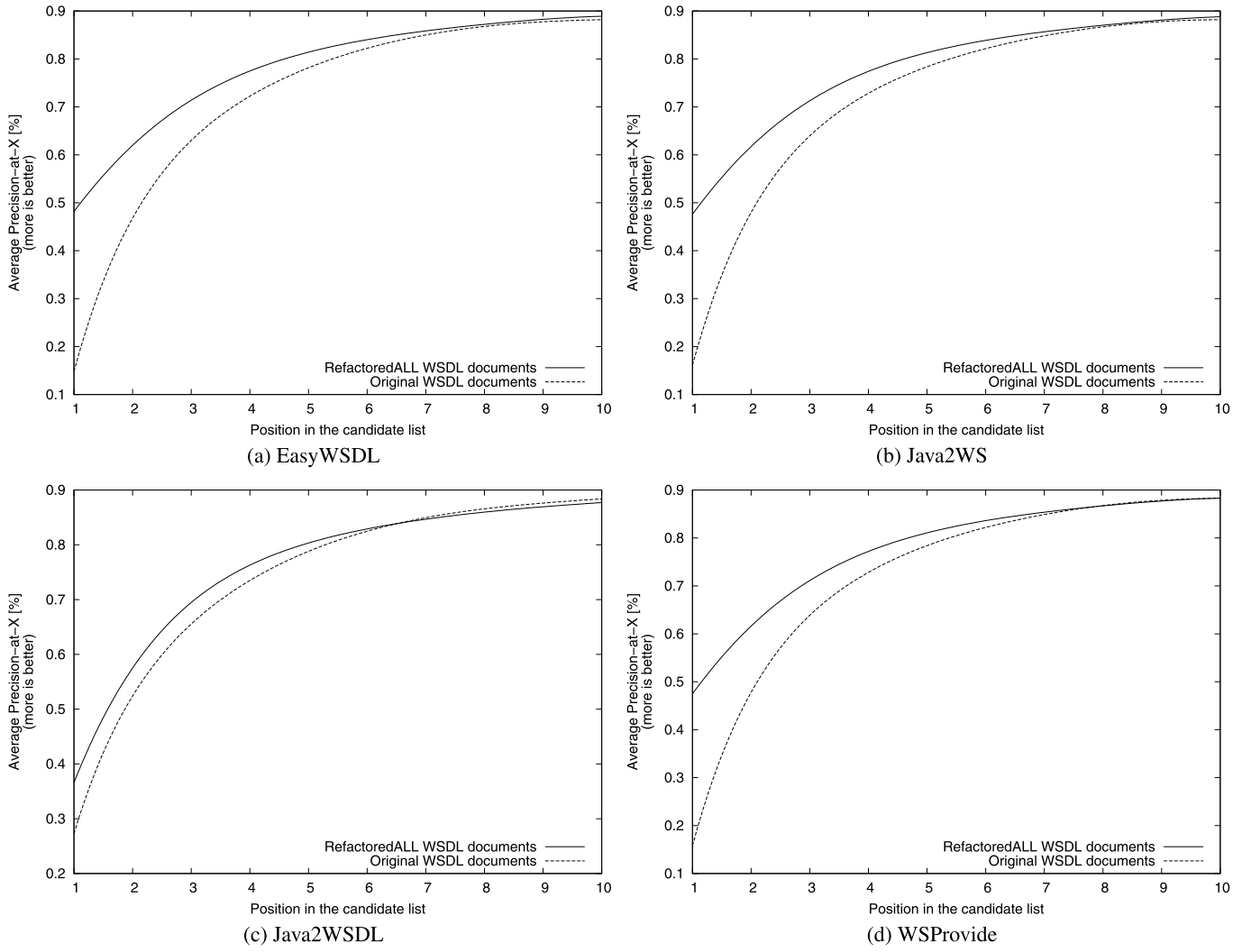


Fig. 22. Impact on retrievability when combining all refactorings and using WSQBE **with** noise.

Table 5
Lucene4WSDL **without** noise data-set Precision-at-1 (top subtable), recall (center subtable) and nDCG (bottom subtable).

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[17.9;53.7]%	[35.4;42.0]%	[36.5;41.0]%	[35.4;42.0]%	[17.1;54.4]%
Java2WS	[25.1;48.6]%	[35.7;41.7]%	[37.0;40.3]%	[35.7;41.7]%	[24.7;48.9]%
Java2WSDL	[27.3;47.0]%	[36.2;42.3]%	[40.3;38.2]%	[36.2;42.3]%	[25.6;48.5]%
WSProvide	[22.1;51.1]%	[34.3;43.7]%	[34.3;43.7]%	[34.0;43.8]%	[21.9;51.0]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[92.9;93.9]%	[94.1;94.3]%	[94.3;94.2]%	[94.1;94.3]%	[92.9;94.2]%
Java2WS	[93.0;93.9]%	[94.4;94.4]%	[94.6;94.4]%	[94.4;94.4]%	[93.0;94.1]%
Java2WSDL	[94.0;94.3]%	[94.8;94.8]%	[95.2;95.1]%	[94.8;94.8]%	[94.0;94.5]%
WSProvide	[92.6;93.4]%	[94.2;94.2]%	[94.4;94.4]%	[94.2;94.2]%	[92.7;93.5]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[81.3;85.1]%	[85.8;86.0]%	[86.0;86.0]%	[85.8;86.0]%	[81.1;85.3]%
Java2WS	[82.8;84.1]%	[85.8;85.8]%	[86.1;85.8]%	[85.8;85.8]%	[82.7;84.3]%
Java2WSDL	[84.0;83.8]%	[86.7;86.7]%	[87.3;86.5]%	[86.7;86.7]%	[83.8;84.1]%
WSProvide	[82.4;83.8]%	[86.1;86.0]%	[86.2;86.1]%	[86.1;86.0]%	[82.2;84.0]%

With respect to the understandability of the resulting WSDL documents, at first the use of the refactoring operation seems to have a positive effect as it completely removes the presence of the Empty messages anti-pattern [24]. Besides, the effort required to apply it is minimal as it only involves adding a new parameter to a subset of the methods in the service interface. However, it was also shown that adding an unrelated parameter to an operation is considered

Table 6
Lucene4WSDL **with** noise data-set Precision-at-1 (top subtable), recall (center subtable) and nDCG (bottom subtable).

Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[15.1;45.9]%	[28.2;35.1]%	[29.7;33.6]%	[28.2;35.1]%	[14.2;46.7]%
Java2WS	[22.8;42.4]%	[30.1;36.4]%	[31.0;34.8]%	[29.0;36.8]%	[22.4;42.8]%
Java2WSDL	[24.1;38.6]%	[28.6;36.1]%	[28.8;35.8]%	[28.6;36.1]%	[23.2;39.6]%
WSProvide	[19.7;45.0]%	[29.9;36.2]%	[30.0;36.1]%	[30.0;36.1]%	[19.2;45.1]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[86.5;88.6]%	[87.5;87.3]%	[87.5;87.3]%	[87.5;87.3]%	[86.4;88.9]%
Java2WS	[88.2;89.2]%	[88.5;88.5]%	[88.5;88.4]%	[88.5;88.5]%	[88.2;89.4]%
Java2WSDL	[87.1;89.4]%	[87.8;88.2]%	[88.0;88.0]%	[87.8;88.2]%	[87.1;89.5]%
WSProvide	[88.0;89.0]%	[88.6;88.8]%	[88.5;88.6]%	[88.6;88.8]%	[87.8;89.2]%
Tool	DS_1	DS_2	DS_3	DS_4	DS_5
EasyWSDL	[73.3;77.7]%	[76.5;76.8]%	[76.6;76.7]%	[76.5;76.8]%	[72.8;78.1]%
Java2WS	[75.9;78.1]%	[78.6;78.7]%	[78.7;78.5]%	[78.6;78.7]%	[75.8;78.3]%
Java2WSDL	[75.1;77.3]%	[77.5;77.7]%	[78.0;77.0]%	[77.5;77.7]%	[74.9;77.7]%
WSProvide	[75.7;77.8]%	[78.7;79.0]%	[78.7;78.8]%	[78.8;78.9]%	[75.5;78.1]%

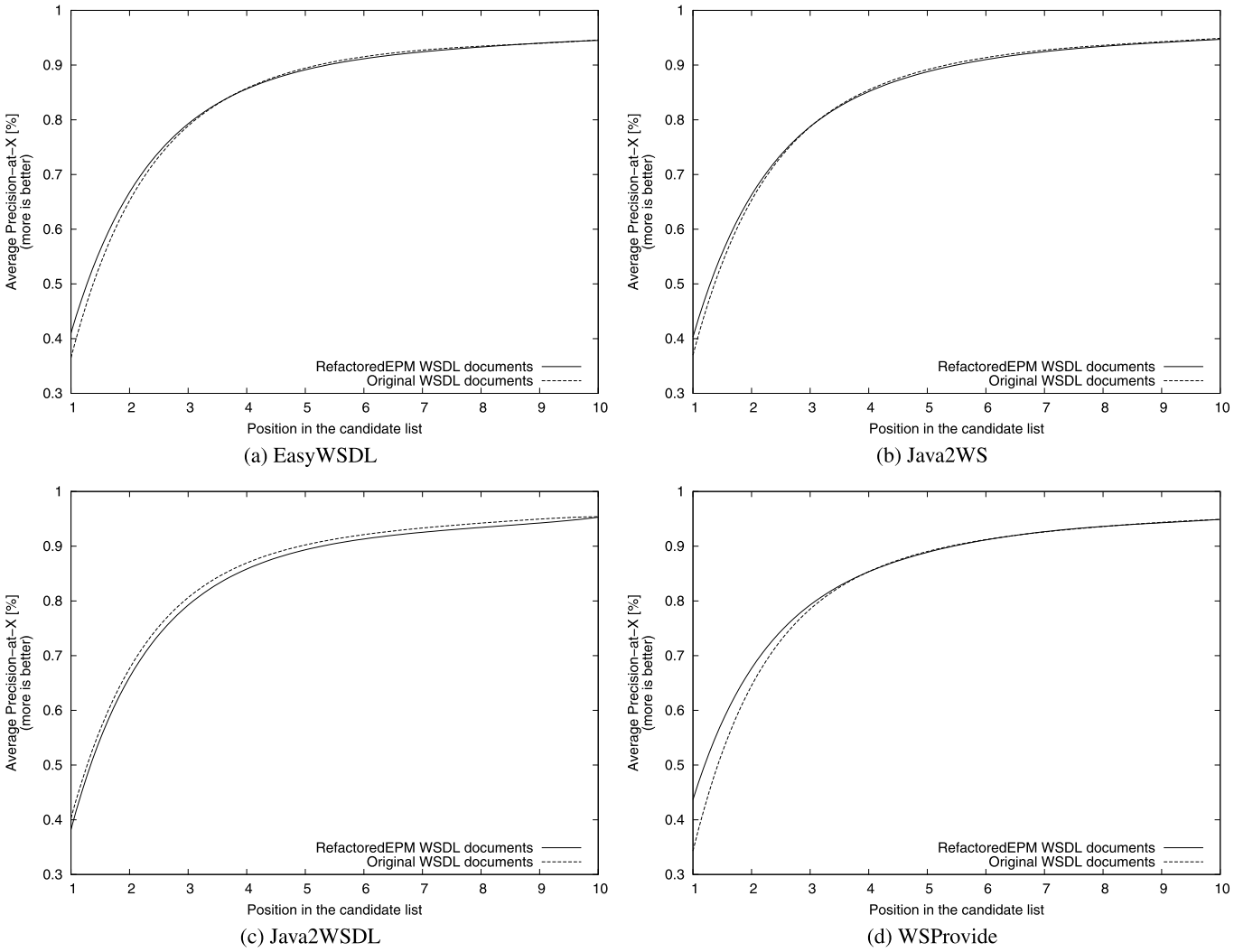


Fig. 23. EPM impact on retrievability when using Lucene **without** noise.

a code smell and should be avoided. Furthermore, an additional input parameter does not help a developer trying to understand a WSDL document as he/she now needs to think about whether or not he should set a value for that parameter to receive the expected response from the invocation to the Web Service. Then, it can be argued that although the

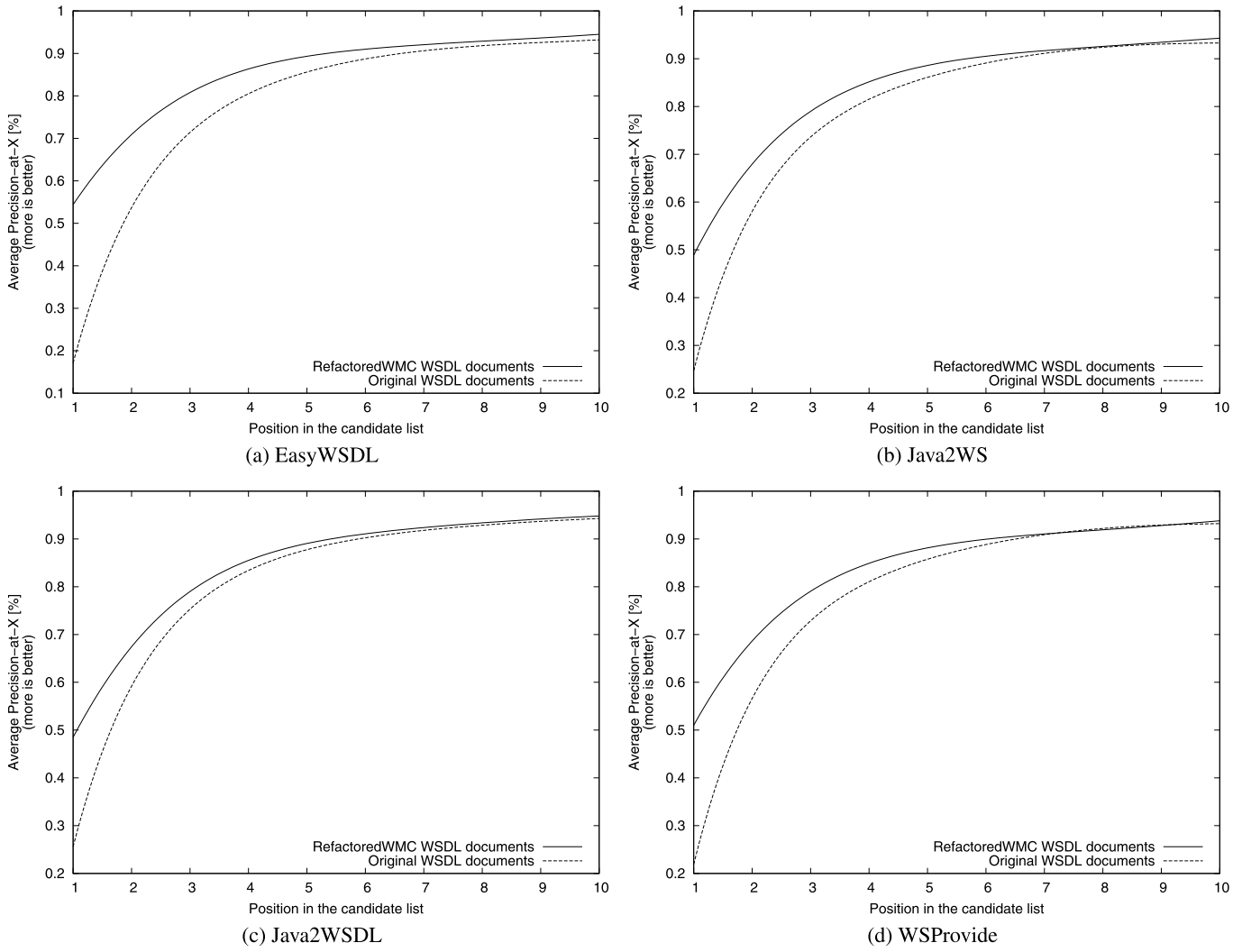


Fig. 24. WMC impact on retrievability when using Lucene **without** noise.

anti-pattern is removed by the refactoring operation, the understandability of the resulting description is not significantly improved.

On the other hand, the results of Sections 4.2 and 4.3 showed that the use of the WMC refactoring operation results in a significant improvement on retrieval with respect to the original data-set. More importantly, similar results were obtained with all the employed experimental setups, making this refactoring operation an appealing choice to drive the code-first code refactorings, since the change on the services code is simple and the positive effect on service retrieval is significant in a real world scenario. Coincidentally, the WMC refactoring was determined to be the most efficient choice in terms of anti-pattern occurrences reduction [24], as it affected all 6 anti-patterns either directly or indirectly. Therefore, while the original WSDL documents contained redundant and non-representative terms, the refactored descriptions were more specific and concise which, as shown in [34], effectively improves the retrieval efficiency of syntactic registries. This suggests that the WMC refactoring operation provides good results both from a discovery and an understandability point of view, provided that common good practices are followed by developers when dividing their services. That is, as described in Section 3.1, to maximize the benefits obtained from this refactoring operation descriptive names should be used for the new interfaces. Similarly, the decision of which operations are exposed by each new service should be made by taking into account how closely related these operations are from a functional cohesiveness point of view.

Despite its clear benefits, the WMC refactoring has some drawbacks in practice when compared to other refactoring operations. Concretely, although the refactoring itself is simple and is automated in most modern IDEs, having two different Web Services instead of one introduces an overhead in service deployment. That is, the effort incurred by service providers to deploy and publish their services is doubled by the simple fact that there are twice as many Web Services after the refactoring is applied. This situation depicts a clear trade-off between the benefits obtained from the refactoring and the effort required to apply it. Besides, when the WMC refactoring is applied aggressively on a service code, apart from causing deployment and performance issues, the same service functionality is scattered across many WSDL documents with few

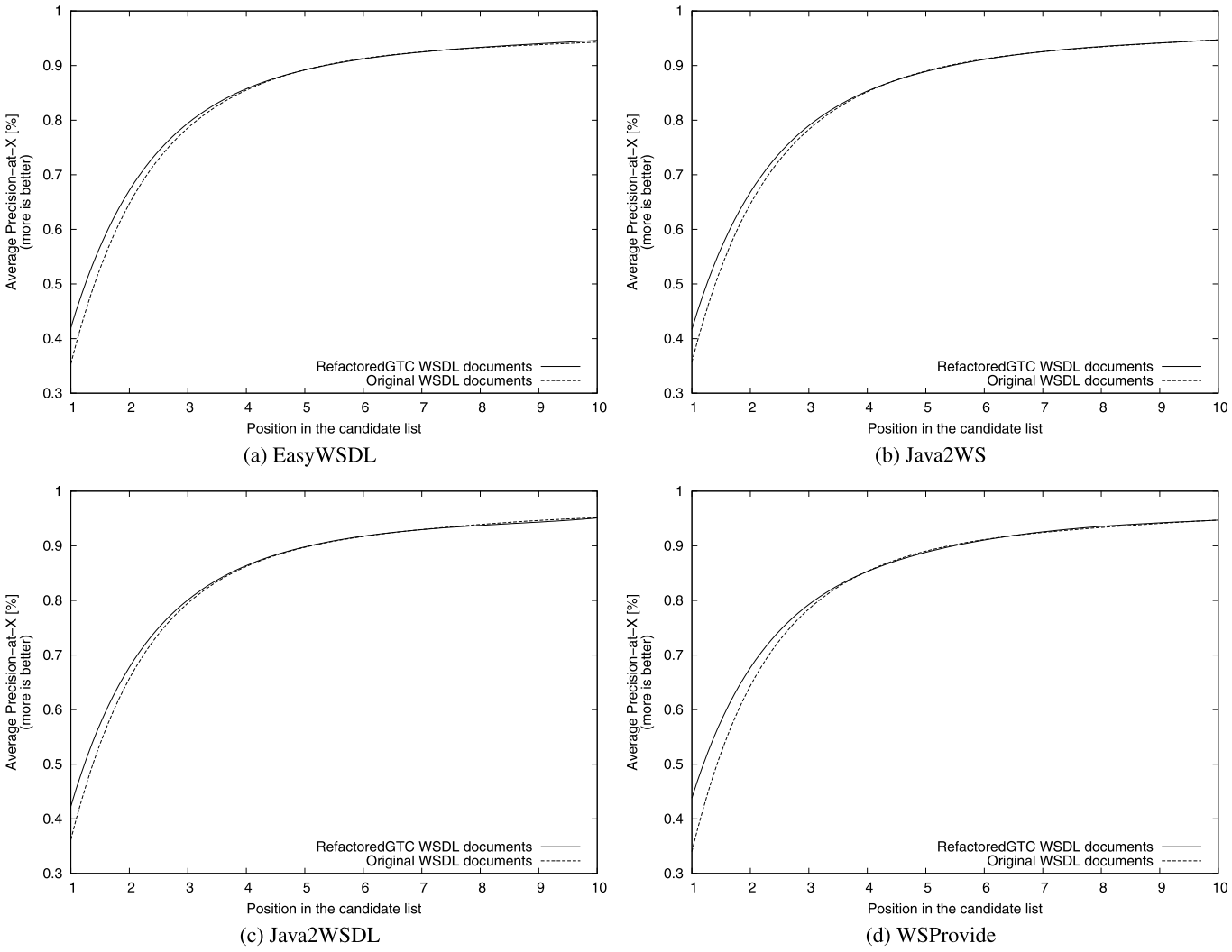


Fig. 25. ATC impact on retrievability when using Lucene **without** noise.

operations. This, in turn, negatively affects understandability, since it is more difficult to get a big picture of the functionality of the service from the many created WSDL documents.

With respect to the ATC refactoring operation, the experimental results showed that the only case when the refactoring results in a significant improvement on service retrieval is when the WSQBE registry is used and only the original and refactored versions of the same data-set are published. However, in a more practical scenario when other WSDL documents are published in the same registry or when the Lucene4WSDL registry is used, the results suggest that the positive effect on retrieval effectiveness is not significant. These results may stem from two facts. First, the Whatever types anti-pattern which is associated with the ATC metric is the least frequent on the generated WSDL documents of the experiments. Then, using this refactoring to remove the presence of the anti-pattern results in a small reduction on the number of total anti-patterns detected. Second, it was shown that the ATC refactoring operation presented trade-off situations where its use produced a decrease of the Whatever types anti-pattern but also an increase on the number of occurrences of the Redundant data models bad practice [24]. Then, while the refactoring has a positive effect on service retrieval because more representative names are being used (i.e., concrete definitions for the data-types exchanged instead of the abstract *xsd:any* construct) it also introduces a negative effect stemming from the use of redundant terms as shown by the increase on the number of occurrences of the Redundant data models anti-pattern.

However, the ATC refactoring results in more legible WSDL documents as developers know exactly the type and structure of the parameters received and returned by Web Services. Furthermore, the effort required to apply the refactoring is not as high as the one incurred by the WMC refactoring as only the interface of the service needs to be changed, resulting in no additional overhead in deployment. Furthermore, this change tends to be fairly simple as it usually involves adding the type of the elements contained in a collection or changing an abstract data-type with a similar but concrete one. But, more effort is required when the classes associated to the new concrete data-types have not been defined. Then, considering that the ATC refactoring does not negatively affect service retrieval, the refactoring operation is an attractive choice for developers since the resulting WSDL documents are easier to understand.

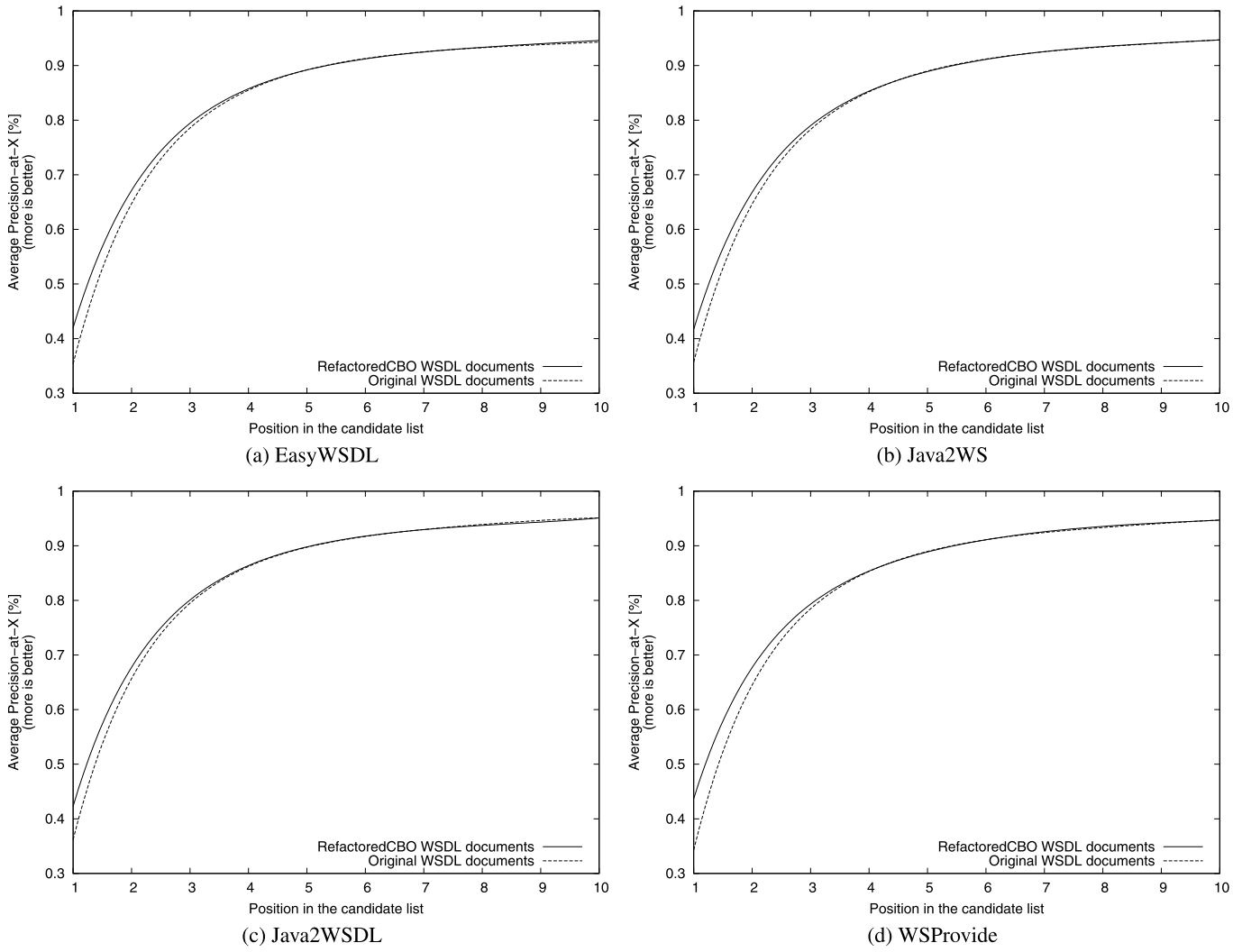


Fig. 26. CBO impact on retrievability when using Lucene **without** noise.

Similarly to the ATC case previously described, the CBO refactoring operation does not result in significant improvements on service retrieval. Concretely, while the refactoring operation results in less verbose WSDL documents because all the exchanged data-types are primitives and therefore they do not need to be declared as XSD data-types on the generated descriptions, this also produces more occurrences of the Redundant data models anti-pattern.

Moreover, the CBO refactoring might result in descriptions that are easier to use by developers because they do not need to worry about building complex domain objects in order to invoke the operations in the Web Services. This translates into greater legibility since developers does not have to clearly understand the (potentially complex) XSD data-types in the descriptions. However, the effort required to apply this refactoring is considerable higher than the one required by ATC as many complex data-types used in services implementations have to be replaced by primitive data-types. This does not only involve heavy changes on service interfaces but sometimes is counterproductive because it incurs in known bad practices such as having long lists of parameters on methods, which in turn negatively affects legibility, and hence it should used sparingly. Unlike the case previously described for the WMC refactoring where the changes were easy to make on the service interfaces but an overhead was added at deployment time, the CBO refactoring does not involve any additional effort in deployment but it causes a considerable overhead at development time.

Finally, although the retrieval efficiency for both registries was improved when all the refactoring operations were applied on the same data-set, this improvement was not as significant as the one obtained when only WMC was considered. Indeed, it was shown that after applying all the refactoring operations on a data-set the resulting number of occurrences of anti-patterns was bigger than the one obtained by refactoring the services focusing only in WMC [24].

Applying all the refactoring operations might results in more legible WSDL documents. Unfortunately, this refactoring is also the more complex to apply. On one hand it suffers from the previously described overhead of the WMC refactoring in service deployment. Additionally, it involves the extra tasks discussed for the ATC and CBO refactorings at development time. Then, similarly to the WMC case discussed previously, the use of all refactoring operations at the same time involves a

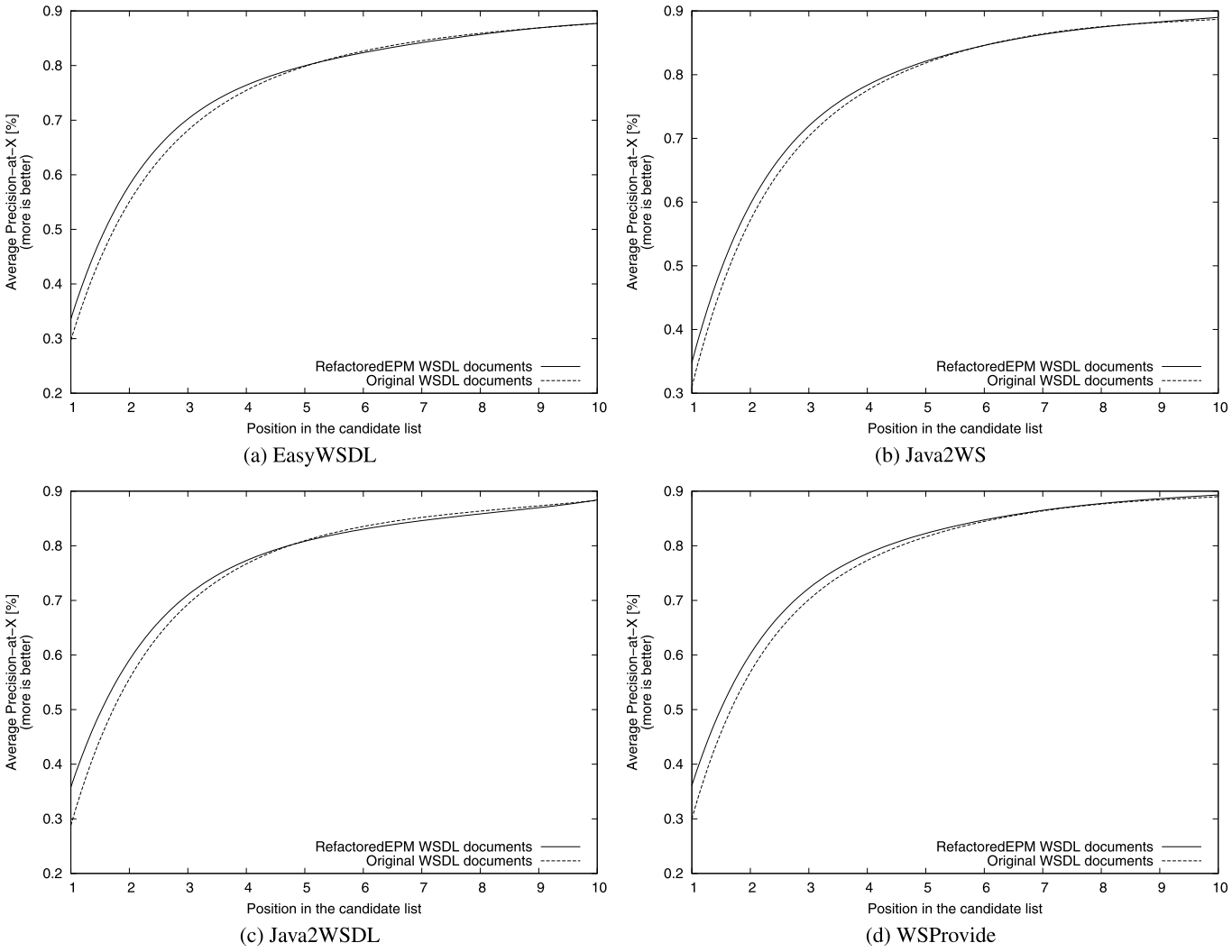


Fig. 27. EPM impact on retrievability when using Lucene **with** noise.

clear trade-off between WSDL documents quality, both in terms of legibility and service retrieval, and a high effort needed to apply the different operations.

It is worth noting that the previous discussion was focused on the Precision-at-1 metric because this latter has the biggest effect on retrieval. As explained, users tend to pay considerably more attention to the first service returned by the registry. The Recall and nDCG metrics, defined in Section 4.1 and whose values were shown in the previous two sections, are global measures that give an overall sense of the registries efficiency when dealing with the original and refactored data-sets. In the former case, it was shown that when the different refactoring operations were applied the Recall metric did not vary its values significantly with respect to the original data-set. That is, the effectiveness of the registries in retrieving relevant documents in a window of *10* documents was not negatively affected by the use of the refactoring operations.

With respect to the nDCG metric, the results show a slight increase on its values when the refactoring operations were applied. Concretely, the metric showed an average increase of 3.3% and 2.72% for the case of the RefactoredWMC and RefactoredAll data-sets, respectively. On the other hand, the RefactoredCBO and RefactoredATC data-sets presented an increase of 0.22% and 0.17%, while the RefactoredEPM data-set resulted in a decrease of the average value of the metric of 0.25%. It is worth noting that these results are highly consistent with that of the Precision-at-1 metric, where the use of the EPM refactoring had a negative impact on the metric, the CBO and ATC refactorings showed a slight improvement and the use of the WMC refactoring operation and all the refactorings applied at the same time showed the most significant improvements, with WMC resulting in the highest values for the Precision-at-1 metric. Furthermore, this positive effect on the values of the metric are more significant in the second set of experiments, when the original and refactored data-sets were published together with the additional data-set of 1664 WSDL documents. Overall, these results suggest that the performance of the registries is improved when the refactoring operations are used, specially in a more realistic scenario, thus providing an appealing practical value.

To conclude this section, Table 7 shows a summary of the benefits and drawbacks of each refactoring operation with respect to the legibility, service retrieval improvements and effort required to apply it. As it is usually the case when

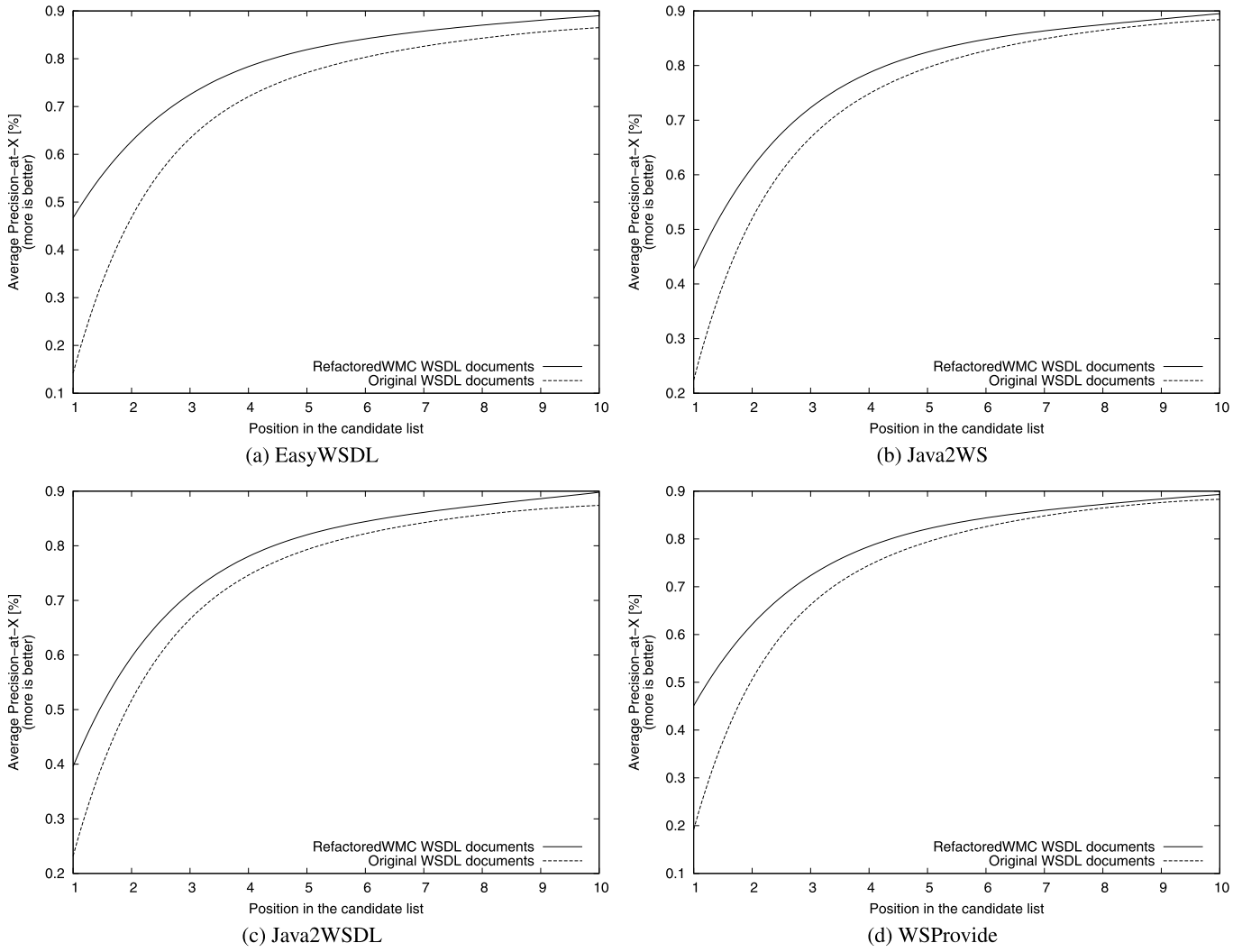


Fig. 28. WMC impact on retrievability when using Lucene **with** noise.

trade-off situations arise, the decision of which refactoring operation to use depends on the main goal followed (i.e., better legibility or better retrieval efficiency) and the effort that developers are willing to invest to achieve that goal.

5. Conclusions and future works

In this paper, the effects of applying some early code refactorings on the implementation of code-first Web Services were deeply analyzed and quantified with respect to the retrieval effectiveness of syntactic registries, WSDL legibility, and involved effort. Each of the five refactoring operations described in previous sections were studied independently by using four different Java to WSDL generation tools and two different registries. The extensive set of experimental data obtained from the experiments performed in this paper provided empirical evidence confirming that the use of the refactoring operations not only results in a decrease on the number of generated WSDL anti-patterns but also in easier to retrieve services, independently of the generation tool used and the registry where they are published. Moreover, the fact that the same experiments were repeated with an additional data-set of 1664 WSDL documents published in the same registry as the original and refactored data-sets, suggests that the refactoring approach can be used in real world scenarios without the need of any additional steps.

It is worth noting that the particular values of the Precision-at-1 metric are different for the four generation tools and the two employed registries. However, this is to be expected given the fundamentally intrinsic differences on how they generate WSDL documents in the former case and how they process a query in the latter. Nevertheless, despite of these minor differences, the general results are consistent across all of them, i.e., a specific set of refactoring operations results in highly significant improvements on service retrievability mostly in terms of Precision-at-*n*, while other refactorings do not have such an acknowledgeable impact but they require less coding effort to be applied.

In this line, the present paper also provided evidence showing that while some refactorings do not have a significant impact on service retrieval they might still be applied by developers to obtain more legible WSDL documents that are easier

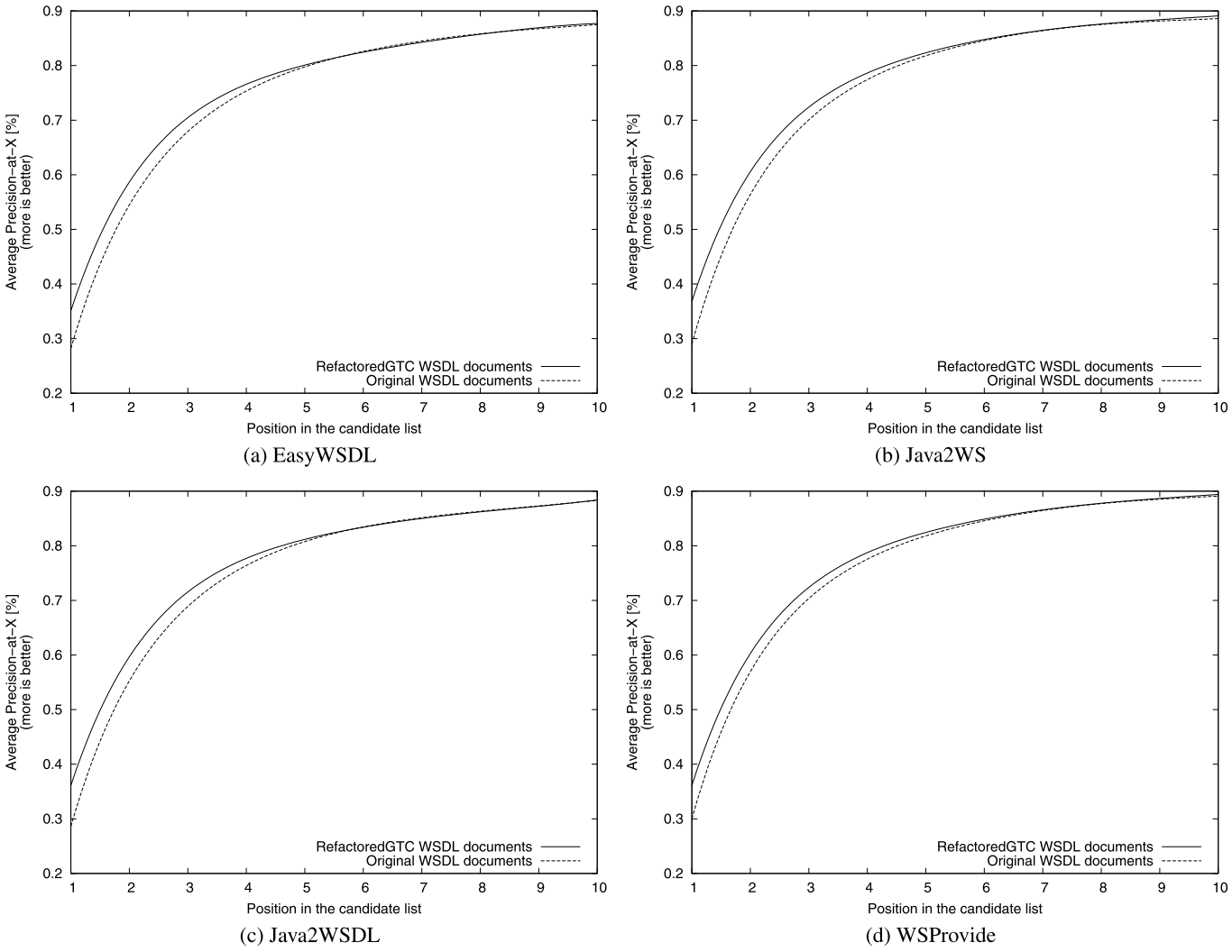


Fig. 29. ATC impact on retrievability when using Lucene **with** noise.

to understand by potential consumers. Moreover, it was shown that each refactoring operation has an associated effort that could lead developers to favor one refactoring over others. In other words, this paper indirectly provides developers with a set of guidelines to help them choose which refactoring operations to use by taking into account three main factors: the legibility of the resulting services descriptions, the retrieval efficiency of syntactic registries, and an indication of the effort required both in development and deployment time needed to apply them. By using these guidelines according to their particular goals, developers can make an informed decision on which refactorings to apply to their services implementations in order to maximize the benefits obtained.

Code-first is a WSDL construction methodology widely employed in the industry because code-first is cheaper and provides a better time to market than contract-first. It seems, however, that contract-first tends to deliver better quality WSDL documents than code-first in terms of discoverability anti-patterns [30,31]. Developers using contract-first have full control over their WSDL documents, and hence most anti-patterns can be removed directly at WSDL specification time. In opposition, correcting possible WSDL anti-pattern causes in a Web Service source code, as in the approach in this paper, does not guarantee that the generated WSDL document will be free of anti-patterns, but less affected by anti-patterns, minimizing their negative impact. For instance, since in Java method returns do not have a name, WSDL generation tools tend to name return messages with generic names, thus suffering from the Ambiguous names anti-pattern. However, if the method signatures do not have ambiguous names, the quantity of ambiguous names in the generated WSDL documents decreases. All in all, our approach does not aim at obtaining optimal WSDL documents in terms of anti-pattern occurrences, but at generating as good quality WSDL documents as possible, while preserving the advantages of the code-first methodology, namely low development cost and fast time-to-market.

In this line, a limitation of the approach is that our refactorings focus on modifying structural aspects of service codes only, such as replacing POJOs by primitive data-types and viceversa, splitting a service into several smaller services, adding/-removing parameters, and so on. However, these refactorings do not cover certain service coding bad practices that lead to textual (but not necessarily structural) legibility and retrievability problems in generated WSDL documents. Examples are

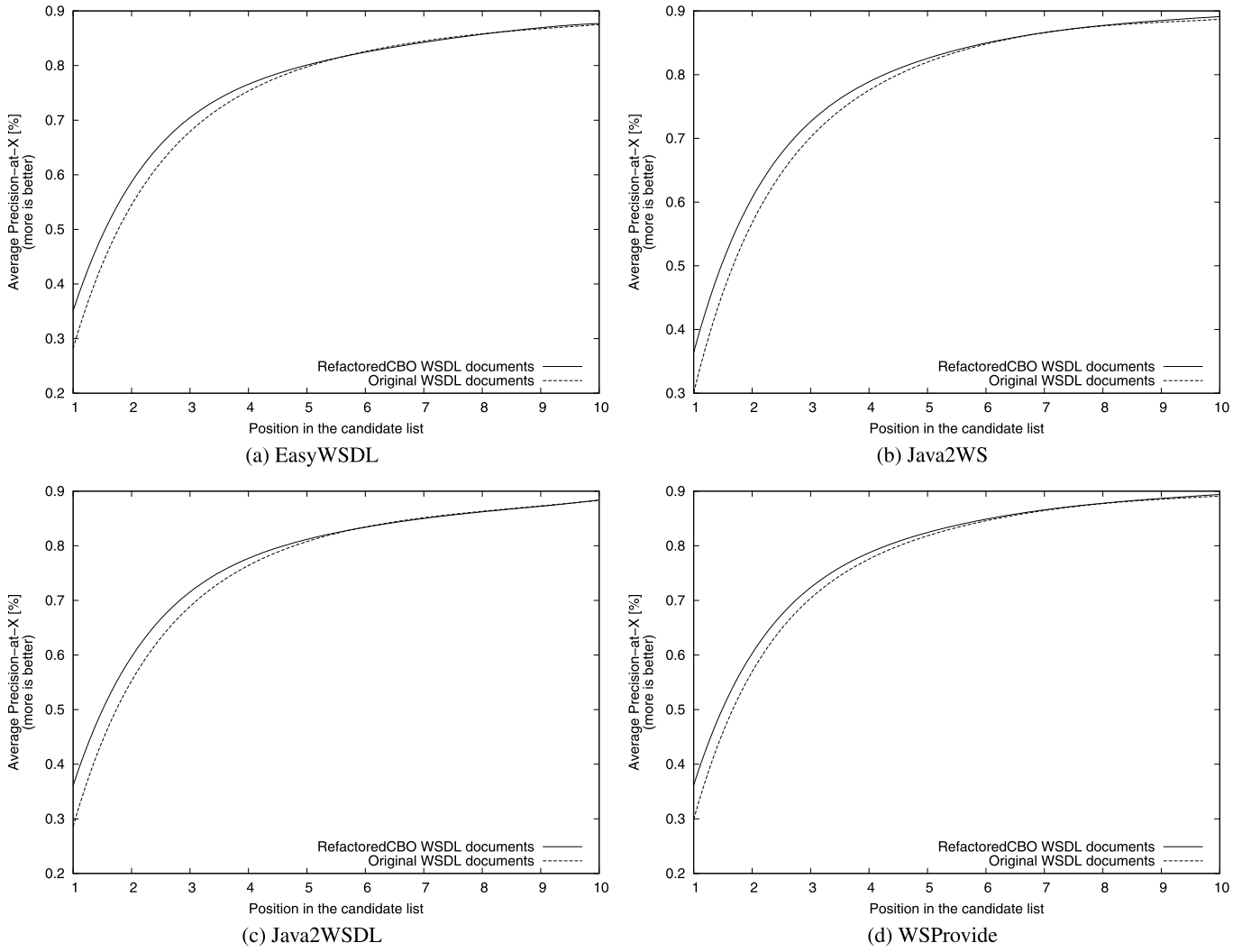


Fig. 30. CBO impact on retrievability when using Lucene **with** noise.

Table 7
Summary of the effects of the refactoring operations.

	EPM	WMC	ATC	CBO	All
Effect on WSDL documents legibility	Low	High	Medium	Medium	High
Effect on service retrieval effectiveness	Low	High	Low	Low	High
Effort needed to apply the refactoring	Low	High	Medium	Medium	High

absence or inappropriate documentation and meaningless/ambiguous parameter and method names. We are investigating source code heuristics to early spot these textual problems, which are based on text mining techniques. The goal is to further increase the quality of obtained code-first WSDL documents. We are backing up this development with similar heuristics already developed in the context of legacy source code to SOA migration [30,31]. This will eventually allow us to better compare the achievable WSDL quality – in terms of WSDL anti-patterns – when using the code-first or the contract-first WSDL construction methodologies.

Through the paper we have focused on tools for generating WSDL documents. However there are cases where these artifacts have been already developed and used. We are extending our WSQBE service registry [11] for mitigating the negative effects of the anti-patterns on discovery. Towards this end we are using techniques for detecting discoverability anti-patterns in WSDL documents [33,32]. If Redundant port-types or Redundant data models anti-patterns are present, the service registry avoids indexing redundant words more than once. On the other hand, if a parameter name is ambiguous, an algorithm analyzes the class of the parameter looking for potential alternative names, which requires the source code of the service implementation. In addition, if the service implementation uses that parameter as a parameter to call another method, the name positionally associated in the latter method to that parameter might provide hints about more suitable names for the ambiguous parameter. We will also explore whether linguistic corpuses, such as WordNet [28] or Wikipedia [22], can be used to narrow down the potential names by reducing a set of words to a synonym or a hypernym.

Another interesting research line is to study the relationships between the anti-patterns and other OO metric catalogs, including traditional metrics such as the ones by Halstead, McCabe, and Henry and Kafura [39], and eventually newer ones [2]. This could in turn lead to investigate the effects of other kind of early refactorings on anti-patterns and service discovery. Then, developers would have a broader range of options when choosing how they refactor their services. Likewise, in the present work, Web Service quality was conceived as a synonym of WSDL understandability and retrievability, and thus the anti-pattern catalog was viewed as a mean to quantify these two aspects from WSDL documents. Alternatively, other authors conceive high quality WSDL documents as those having little complexity [36] or exposing high levels of maintainability [5]. Therefore, these different aspects of Web Service quality should also be considered in future studies. We have made some progress on rethinking refactorings for considering these catalogs [23], but further research is needed.

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions to improve the paper. We also acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PICT-2012-0045.

References

[1] E. Agichtein, E. Brill, S. Dumais, R. Ragno, Learning user interaction models for predicting Web search result preferences, in: 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 2006, pp. 3–10.
[2] J. Al Dallal, Measuring the discriminative power of object-oriented class cohesion metrics, IEEE Trans. Softw. Eng. 37 (6) (2011) 788–804.
[3] E. Al-Masri, Q.H. Mahmoud, Discovering the best web service, in: Proceedings of the 16th International Conference on World Wide Web, WWW '07, ACM, New York, NY, USA, 2007, pp. 1257–1258.
[4] Apache Software Foundation, Java to WS, 2013, http://cxf.apache.org/docs/java-to-ws.html (last accessed April 2013).
[5] D. Baski, S. Misra, Metrics suite for maintainability of extensible markup language Web Services, IET Softw. 5 (3) (2011) 320–341.
[6] J. Beaton, J. Sae Young, X. Yingyu, J. Stylos, B.A. Myers, Usability challenges for enterprise service-oriented architecture apis, in: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 193–196.
[7] M.B. Blake, M.F. Nowlan, Taming Web Services from the wild, IEEE Internet Comput. 12 (5) (2008) 62–69.
[8] J. Bloch, How to design a good api and why it matters, in: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 506–507.
[9] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.
[10] M. Crasso, J.M. Rodriguez, A. Zunino, M. Campo, Revising wsdl documents: why and how, IEEE Internet Comput. 14 (2010) 48–56.
[11] M. Crasso, A. Zunino, M. Campo, Easy web service discovery: a query-by-example approach, Sci. Comput. Program. 71 (2) (2008) 144–164.
[12] M. Crasso, A. Zunino, M. Campo, Combining query-by-example and query expansion for simplifying Web Service discovery, Inf. Syst. Front. 13 (2011) 407–428.
[13] M. Crasso, A. Zunino, M. Campo, A survey of approaches to Web Service discovery in service-oriented architectures, J. Database Manag. 22 (2011) 103–134.
[14] T. Erl, SOA Principles of Service Design, Prentice Hall, 2007.
[15] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.
[16] B. Huston, The effects of design pattern application on metric scores, J. Syst. Softw. 58 (3) (2001) 261–269.
[17] JBoss Community, JBossWS – wsprovide, 2012, https://community.jboss.org/wiki/JBossWS-Wsprovide (last accessed April 2013).
[18] R.R. Korfhage, Information Storage and Retrieval, John Wiley & Sons, 1997.
[19] C. Mateos, M. Crasso, A. Zunino, J.L. Ordiales Coscia, Avoiding wsdl bad practices in code-first web services, SADIO Electron. J. Inform. Oper. Res. 11 (1) (2012) 31–48, Special issue dedicated to ASSE 2011, http://sadio.opentierra.com/SADIO-Files/Avoiding_WSDL.pdf.
[20] C. Mateos, M. Crasso, A. Zunino, J.L. Ordiales Coscia, Revising wsdl documents: why and how – part ii, IEEE Internet Comput. 17 (5) (2013) 46–53.
[21] M. McCandless, E. Hatcher, O. Gospodnetic, Lucene in Action, second edition: covers Apache Lucene 3.0, Manning Publications Co., Greenwich, CT, USA, 2010.
[22] D. Milne, I.H. Witten, An open-source toolkit for mining Wikipedia, Artif. Intell. 194 (0) (2013) 222–239.
[23] J.L. Ordiales Coscia, M. Crasso, C. Mateos, A. Zunino, Estimating web service interface quality through conventional object-oriented metrics, CLEI Electron. J. 16 (1) (2012), Special Issue CibSe/ESELAW.
[24] J.L. Ordiales Coscia, C. Mateos, M. Crasso, A. Zunino, Anti-pattern free code-first Web Services for state-of-the-art Java WSDL generation tools, Int. J. Web Grid Serv. 9 (2) (2013) 107–126.
[25] OW2 Consortium, EasyWSDL toolbox, 2010, http://easywsdl.ow2.org/ (last accessed April 2013).
[26] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: a research roadmap, Int. J. Coop. Inf. Syst. 17 (2) (2008) 223–255.
[27] J. Pasley, Avoid XML schema wildcards for Web Service interfaces, IEEE Internet Comput. 10 (2006) 72–79.
[28] J.S.R. Feldman, The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data, Cambridge University Press, 2006.
[29] J. Ramey, T. Boren, E. Cuddihy, J. Dumas, Z. Guan, M.J. van den Haak, M.D.T. De Jong, Does think aloud work? How do we know?, in: CHI '06 Extended Abstracts on Human Factors in Computing Systems, CHI '06, ACM, New York, NY, USA, 2006, pp. 45–48.
[30] J. Rodriguez, M. Crasso, C. Mateos, A. Zunino, M. Campo, Bottom-up and top-down cobol system migration to web services: an experience report, IEEE Internet Comput. 17 (2) (2013) 44–51.
[31] J.M. Rodriguez, M. Crasso, C. Mateos, A. Zunino, M. Campo, G. Salvatierra, The SOA frontier: experiences with three migration approaches, in: Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments, IGI Global, 2013, pp. 126–152.
[32] J.M. Rodriguez, M. Crasso, A. Zunino, An approach for web service discoverability anti-patterns detection, J. Web Eng. 12 (1–2) (2013) 131–158.
[33] J.M. Rodriguez, M. Crasso, A. Zunino, M. Campo, Automatically detecting opportunities for web service descriptions improvement, in: W. Cellary, E. Estevez (Eds.), Software Services for e-World, in: IFIP Advances in Information and Communication Technology, vol. 341, Springer, Berlin, Heidelberg, 2010, pp. 139–150, http://dx.doi.org/10.1007/978-3-642-16283-1_18.
[34] J.M. Rodriguez, M. Crasso, A. Zunino, M. Campo, Improving web service descriptions for effective service discovery, Sci. Comput. Program. 75 (11) (2010) 1001–1021.
[35] G. Salton, A. Wong, C.S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620.
[36] H.M. Sneed, Measuring Web Service interfaces, in: 12th IEEE International Symposium on Web Systems Evolution (WSE) 2010, 2010, pp. 111–115.
[37] D. Spinellis, Tool writing: a forgotten art?, IEEE Softw. 22 (2005) 9–11.
[38] The Apache Software Foundation, Java2WSDL: building WSDL from Java, 2005, http://ws.apache.org/axis/java/user-guide.html#Java2WSDLBuildingWSDLFromJava (last accessed April 2013).
[39] F.F. Tsui, O. Karam, Essentials of Software Engineering, Prentice Hall, 2006.