This article is a pre-print of the article "H. Hirsch, A. Rodriguez, J. M. Rodriguez, C. Mateos and A. Zunino: "A Tool for Building Retrievable Code-first Web Services". Proceedings of the 2014 ARGENCON, pp. 165-170. 2014."
The published version is available at http://dx.doi.org/10.1109/ARGENCON.2014.6868490

# A tool for building indexable and searchable code-first Web Services

Matías Hirsch*[1], Ana Rodriguez*[2], Juan Manuel Rodriguez*[3], Cristian Mateos*[4], Alejandro Zunino*[5] and José Luis Ordiales Coscia†[6]

*ISISTAN, UNICEN-CONICET Campus Universitario - Paraje Arroyo Seco, Tandil, Buenos Aires , Argentina
†ThoughtWorks Porto Alegre, Rio Grande do Sul, Brazil
[1]matias.hirsch@isistan.unicen.edu.ar
[2]ana.rodriguez@isistan.unicen.edu.ar
[3]juanmanuel.rodriguez@isistan.unicen.edu.ar
[4]cristian.mateos@isistan.unicen.edu.ar
[5]alejandro.zunino@isistan.unicen.edu.ar
[6]jlordiales@gmail.com

***Resumen*—Although high quality WSDL documents are a key factor in a Web Service system success, previous works have shown that WSDL document quality, in terms of readability and discoverability, is often disregarded by service developers. One of the main causes of this is that developers do not write the WSDL documents directly; instead these latter are automatically generated from service implementations, i.e. code-first Web Services. This work presents a novel tool for assisting Java code-first Web Service development that spot potential issues that might lead to low quality WSDL documents. The approach was empirically evaluated using 81 open-source Web Service implementations. The evaluation shows that the WSDL documents generated using our tool are more easily indexed in and searched from Web Service registries. This indirectly implies that these WSDL documents have a better quality than the ones generated using conventional tools.**

***Resumen*—A pesar de que la calidad de los documentos WSDL, en términos de legibilidad y facilidad de su descubrimiento, es un factor clave en el éxito de los sistemas orientados a servicios, trabajos anteriores han mostrado que la calidad de éstos tiende a ser baja en la práctica. Una posible causa es que los documentos no son escritos manualmente sino generados automáticamente a partir del código de los servicios. En este trabajo se presenta una herramienta para asistir al desarrollo de Servicios Web desarrollados en Java mediante la metodología *code-first*, en donde los documentos WSDL son derivados de la implementación del servicio. La herramienta detecta potenciales problemas en código fuente que pueden afectar la calidad de los documentos WSDL generados. La herramienta fue evaluada utilizando 81 implementaciones de Servicios Web de código abierto. La evaluación mostró que los documentos WSDL generados aplicando las mejoras propuestas por la herramienta son más fáciles de indexar y buscar en registros de Servicios Web. Esto sugiere que la calidad de estos documentos WSDL es mayor respecto de aquellos generados mediante herramientas convencionales.**

## I. Introduction

Web Services is the common technological choice for implementing remote services [1]. Basically, Web Services enable service providers to implement their services using well-known interoperable Web protocols, such as HTTP or SOAP. In this context, services are offered using sets of atomic operations that are described using the Web Service Description Language (WSDL). Hence, service consumers do not need to know service implementation details because knowing the WSDL document associated to a service is in principle enough for determining the functional capabilities of a service. In addition, WSDL documents are used by service registries to index the services [2], which can be instead inspected by consumers usually via keyword-based search interfaces.

Unfortunately, in practice, most developers disregard WSDL document quality. This results in WSDL documents that are difficult to be understood by the service consumers. Furthermore, service registries effectiveness is negatively affected by low quality WSDL documents [3]. In addition, WSDL documents can be a decisive factor in the success of Web Service based enterprise systems [4]. As a result of this, a tool for detecting issues in WSDL documents has been already proposed [5], which is useful in scenarios where WSDL documents are written manually. This is known as contract-first.

However, since the most popular approach to build Web Services in the industry is code-first, the proposed tools and guidelines for improving WSDL document quality [3], [5] are inappropriate. This is because under code-first development, WSDL documents are not written by developers, but instead they are automatically derived from the service implementation. Yet, there is evidence that developers might improve generated WSDL documents quality by first improving the service implementation quality [6].

This paper presents a novel approach that aims at detecting four coding practices which might lead to bad quality WSDL documents. This approach was evaluated using a data-set of 81 code-first Web Service implementations. Besides, we used three WSDL document generation tools to set the basis for comparison. Two of the tools are the well-known Axis[1] and EasyWSDL[2], which are very popular in the industry, while the third is our tool, called Gapidt, which

[1]http://axis.apache.org/axis/
[2]http://easywsdl.ow2.org/

was designed to further reduce the above mentioned issues in generated WSDL documents.

The rest of the paper is organized as follows. Section II presents further details of the problem and related works. Section III describes our tool for detecting problems and refactoring code-first services. Section IV discuses the empirical assessment of the tool. Section V concludes the paper.

## II. Background

The notion of WSDL quality in this work is based on the absence of eight anti-patterns presented in [7]. These anti-patterns are mainly related to the quality of textual/-structural information present in the WSDL documents, and particularly data-types and operations definitions. Regarding textual information, the anti-patterns describe commonly found bad practices in operation elements and comments. For instance, *param1* is not a good name for an operation parameter, and operations must have a textual comment associated. In regard to the WSDL document structure, the anti-pattern catalog states that data-types should be as specific as possible, and operations in a service must have functional cohesion. Finally, the WSDL documents should neither define redundant data-types nor redundant operations.

WSDL document quality is essential in the success of service-oriented systems because WSDL documents are a key component to effectively consume Web Services. When the WSDL documents have low quality, the service consumers cannot easily understand how to use the services. This is true even when the Web Services are intended to be used in an enterprise intranet, i.e., they are intended to be used by developers of the same organization. An example of this is described in [4], in which an enterprise system was migrated to a service-oriented system. However, in the first migration attempt developers disregarded WSDL document quality. This forced a second migration attempt to obtain a set of services that could be easily re-used. In addition, a tool was developed to assist this kind of migrations [8].

Since WSDL document quality is a key factor in service oriented systems, researchers have developed methodologies that aim at improving WSDL document quality [6], [9], [5]. In particular, the tool presented in [5] detects potential issues in WSDL document by means of WSDL document syntax analysis and natural language processing techniques. Although this tool is effective, it is ineffective when using code-first development because it requires manually writing the WSDL documents. In contrast, Ordiales et al. [6] propose a methodology for code-first service development, which shows that there is a statistical correlation between some traditional Object-Oriented (OO) metrics [10] in the service implementation and the number of anti-patterns [7] in the generated WSDL documents. However, the approach is not able to remove text-related anti-patterns.

Instead of improving the code using traditional OO metrics to generate better WSDL documents [6], this work proposes to detect the issues that are more likely to result in WSDL anti-patterns in the service soure code . To do so, we have adapted some of the techniques proposed in [5] to detect the issues in Java service implementation. For instance, the proposed tool is able to analyze the comments in Java code and contrast them to the method signature to ensure that they are cohesive, e.g., a method comment that only contains the class author information is not useful to service consumers.

## III. Approach to Anti-pattern Avoidance

The goal of our approach is twofold: designing a WSDL document generation tool that considers as much textual and structural information as possible and a tool that detects implementation issues that might lead to low quality WSDL documents. The generation tool, called Good Api Design Tools (Gapidt), works very similar to generation tools like Axis or EasyWSDL, but considers more information than these tools, e.g., it uses comments in Java methods to comment service operations.

Gapidt consists in techniques for detecting anti-patterns in WSDL documents [5] adapted to Java source codes and some complementary new issue detection techniques. The tool analyzes the classes that define a service operations, e.g., the classes that would be used to generate the WSDL document by any conventional generation tool. We call these classes *facade classes*. Basically, Gapidt detects the following issues: *Inappropriate or lacking comments*, *Ambiguous names*, *Low cohesive operations*, *Generic return or parameter types* and *Piggybacking errors in return types*. Solving these issues target the anti-patterns described in Table I. Notice that there is a one-to-one relationship between anti-patterns and issues. This tool has been implemented as an Eclipse plug-in[3].

Cuadro I
Targeted WSDL anti-patterns

| Anti-pattern | Issue |
|---|---|
| Inappropriate or lacking comments | Occurs when: a WSDL document has no comments, or comments are inappropriate and non explanatory. |
| Ambiguous names | Occurs when ambiguous or meaningless names are used for denoting the main elements of a WSDL document. |
| Low cohesive operations | Occurs when port-types have weak semantic cohesion. A port-type is a set of operations in a service. |
| Generic return or parameter types | Occurs when a special data-type is used for representing any object of the problem domain. |
| Piggybacking errors in return types | Occurs when operation returns are used to notify service errors. |

To solve *Inappropriate or lacking comments*, the tool firstly verifies that all methods in the facade class have comments, and when some method lacks comments a problem is reported to the developer. Then, the tool analyses the existing comments and compares them with the method name and method parameter names using a similar algorithm to the one presented in [5]. To perform this analysis, the tool generates two hypernym trees, one for verbs and one for nouns for the method comment, and two others for the method name and method parameter names. Finally, the tool determines the similarity between the comment verb trees and method verb trees as well as the similarity between

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


```
SERVICE IMPLEMENTATION

public class Parking{
  public void parkCar(Car c, Location l){...}

  public Location locateCar(String license){...}

  public long elapsedHours(Date a, Date b){...}
}
```

```
public class Car{
   String owner;
   String license;
   ...
}
```

```
public class Location{
   int floor;
   int place;
   ...
}
```

Figura 1. Cohesion graph

comment noun trees and method noun trees. If the similarity is less than 10 %, the tool reports the associated issue. This threshold has been proved to be the most effective value [5]. The trees are constructed as follows:

1. Tag words in the text using a probabilistic context free grammar (PCFG) parser [11], then extract two sets: nouns and verbs.
2. For each set, obtain all the word hypernyms using WordNet [12], e.g., transforms the words sets in hypernyms sets, which are organized from the most generic to the most specific word.
3. For each hypernym set, combine all the elements into a tree.
   *a*) The tree starts with one ROOT node that is not associated to any concept.
   *b*) For each hypernym list; for each element in the list
      1) if the element is in the tree do nothing
      2) else, add the element as child of its hypernym; if it has no hypernym, add it as child of the ROOT node.

Then, the similarity between two trees is computed as follows:

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

For detecting *Ambiguous names*, the tool applies a three-step heuristic. First, it verifies whether the names of classes, attributes and method arguments have an appropriate length, which should fall between 3 and 30 characters [7]. Then, the names are contrasted against a known unrepresentative names list: *param, arg, var, obj, object, foo, input, output, in#, out# and str#*, where # might be a number or nothing. In the third step the names are analyzed using the PCFG, and names of parameters are expected to be "nouns", while the operation names are expected to be "verb+noun". To improve the PCFG in the case of operation names, the tool adds "it must" at the beginning of the names.

Third, to detect *Low cohesive operations*, the tool creates an undirected graph that represents the relations of methods in a facade class [13]. An example of a cohesion graph is depicted in Figure 1. The graph has three kinds of nodes: methods, nouns in method names and classes. In the last case, primitive-types, and Object and String classes are enhanced with their parameter name. The nodes are connected by the following kind of edges:

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

Once the graph including the methods of a facade class has been generated, the tool determines whether the graph is connected or not. Otherwise, the tool selects the largest connected subgraph and reports methods that are not in this subgraph as non cohesive (e.g., the method elapsedHours in Figure 1).

To detect *Generic return or parameter type*s, the tool uses a known list of too generic classes [14]. If a method uses one of these classes, the tool reports the issue. The classes in this list are Object, Vector, List, Map, Collection, Enumeration, Vector<Object>, List<Object>, Map<Object, Object>, Collection<Object> and Enumeration<Object>.

Finally, detecting *Piggybacking errors in return types* involves analyzing the structure of a method output. Firstly, the tool verifies whether a method has exceptions defined. If this is not the case, the tool analyses the output class field names looking for the following keywords: "ping", "error", "errors", "fault", "faults", "fail", "fails", "exception", "exceptions", "overflow", "mistake", "misplay". These names often indicate that the output conveys error information that should be returned using an exception instead of placing the information in the return value [7], [5].

## IV. Empirical Evaluation

To evaluate the effectiveness of the different issue detection techniques, we employed 81 open source Java service implementations (facade classes) exposing 637 methods as service operations. For each detected issue, we refactored the Java projects to remove all the problems, and then we generate the WSDL documents using Gapidt, i.e., we generated a separate WSDL document set for each issue. Since the *Piggybacking errors in return types* issue was not present in the operations, we did not generated a WSDL documents set for this issue. In addition, we generated two sets of WSDL documents from the original Java projects using Axis and EasyWSDL.

For evaluating how the issues impact on WSDL document searchability, we deployed several instances of a service registry called WSQBE [15]. This is, for each issue, a WSQBE was deployed. Each WSQBE instance was fed with a WSDL document set obtained from the original projects affected by the issue, i.e., the WSDL documents generated with Axis or EasyWSDL, one of the WSDL document sets obtained from the refactored projects, and a random WSDL document set that acts as noise. Notice that the number of WSDL documents indexed was different for each issue.

| Anti-pattern |
|---|
| Ambiguous names |
| Low cohesive operations |
| Generic return or parameter types |
| Lacking or inappropriate comments |
| **WSDL Tools** |
| Axis vs Gapidt |
| EasyWSDL vs Gapidt |
| **Noise level** |
| Noise 1 (393 WSDL documents) |
| Noise 2 (1,000 WSDL documents) |
| Noise 3 (1,664 WSDL documents) |
| Noise 4 (2,047 WSDL documents) |

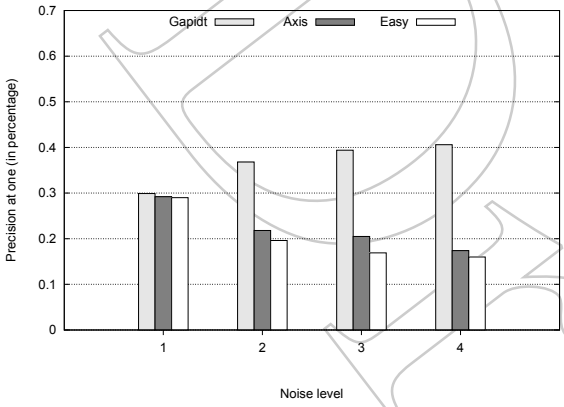


Figura 2. Inappropriate or lacking comments experiment

For example, the WSQBE instance of the *Ambiguous names* issue indexed 44 WSDL documents plus the noise WSDL documents, while the number of WSDL documents involved in the WSQBE instance of the *Generic return or parameter types* issue was 7.

Using the WSQBE registry, we assessed the percentage of queries that retrieved first the associated WSDL document. This means that the WSDL document that was generated using the Java class which is being used as the query, in the first position of the list, metric known as precision-at-1. The experimentation methodology was firstly proposed in [6] and aimed at determining whether a registry ranks first a WSDL document from the original set or from the refactored ones. Their experiments have naturally shown that the lowest accumulative precision is when a rank window of one is taken, and rapidly grows towards 100 % when the window size grows. However, people using search engines tend to disregard results that are ranked after the third and sometimes even the second position in the ranking [16]. Then, we only report precision-at-1 results. Table II depicts the experimental variables where each scenario is defined as a combination of an anti-pattern, a pair of WSDL tools and a noise level.

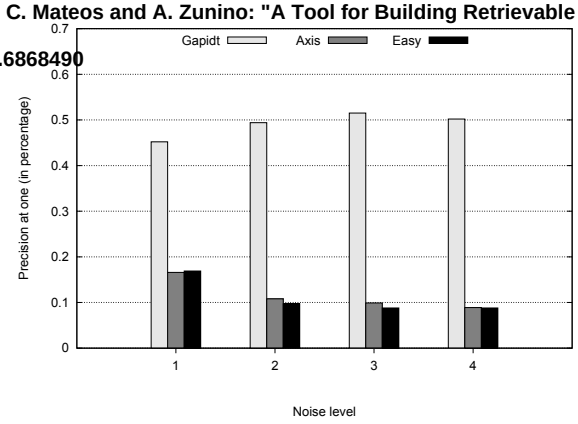


Figura 3. Ambiguous names experiment

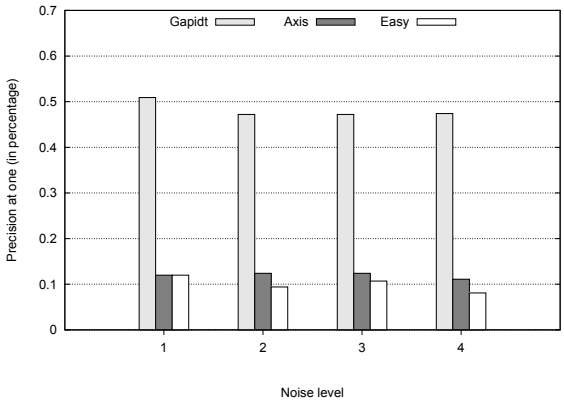


Figura 4. Low cohesive methods experiment

For *Inappropriate or lacking comments,* the tool detected that 503 out of the 637 analyzed methods were affected. These methods were in 78 of the projects, which means that 3 projects were not affected by this issue. Figure 2 presents the improvements in the search results when removing this issue. Although this strategy did not present improvements in the Noise 1 scenario, the precision-at-1 was doubled in the scenarios with more noise. This seems to indicate that correcting this issue is important for large registries with a lower proportion of relevant services.

According to our tool, *Ambiguous names* affected 303 methods in 44 Web Services. Solving this issue in the service implementation resulted in a great improvement in the precision-at-1 (see Figure 3). When indexing the WSDL documents generated from the original projects, WSQBE had a precision-at-1 between 0.1 and 0.2. In contrast, when indexing the WSDL documents of the refactored projects, the precision-at-1 increased up to 0.52 and it was no lower than 0.45.

With regard to *Low cohesive operations* (see Figure 4), our tool detected 44 methods in 20 projects that are not related to the main facade class functionality. This kind of methods that are translated into operations negatively impacts on service registries because they introduce irrelevant terms into the service description, which is poorly indexed instead. Furthermore, these operations might mislead service consumers resulting in difficulties to use the services [7]. According to the empirical evaluation, the WSQBE

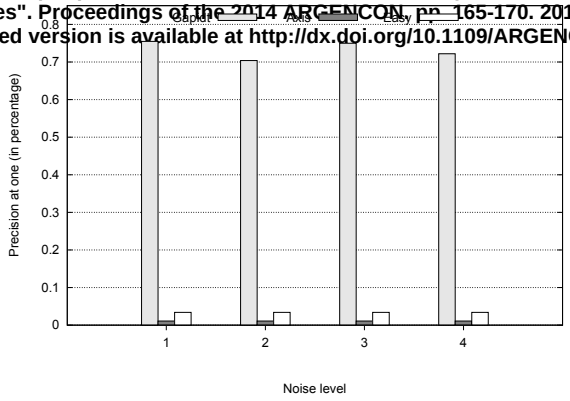


Figura 5. Generic return or parameter types experiment

precision-at-1 increased from an approximated value of 0.1 to 0.5 in all the experimental scenarios.

Finally, refactoring the detected occurrences of *Generic return or parameter types* results in the greatest improvement in the precision-at-1. As it is depicted in Figure 5, the precision switches from less than 0.1 to more than 0.68. This is notable because only 23 methods in 7 facade classes were affected. However, using generic types introduce many irrelevant terms to WSQBE and results in WSDL documents poorly represented, which affects negatively the performance of the registry.

All in all refactoring the detected issues resulted in WSDL documents that are better retrieved by the service registry. In this context, we can ensure that solving the issues improves the WSDL documents quality making them easier to be found. This is important because Web Services are meant to provide services for third-parties that search for the functionality they need.

## V. Conclusions

This work presented a novel tool that assists code-first Web Service developers to write Java service implementations that are mapped to high quality WSDL documents. As claimed in a previous work, this is essential for the success of Web Service systems [4]. Although there are some tools pursuing similar goals [6], [5], [8], as far as we know, there are not tools aiming at directly solving the issues that lead to all WSDL anti-patterns in code-first services. According to our empirical evaluation, using our tool to refactor the service implementation results in WSDL documents that are easy to be found in service registries. In this context, the WSDL documents that are more easily found are considered to have better quality.

In future work, we plan to further evaluate the WSDL documents obtained using the refactored services to effectively assess to which degree the incidence of each anti-pattern is reduced, for example by using human queries. In addition, we plan to generate the WSDL documents from the refactored code using Axis and EasyWSDL. This is because some anti-patterns are actually introduced by these tools and not by developers. For instance, Axis discards source code comments, therefore the generated WSDL documents will lack comments even when the service implementation is commented.

Finally, we will analyze the impact of our tool in other search engines, such as Lucene4WS. Furthermore, we will apply all the refactorings at once to assess their combined effect on service search. Finally, we will consider other information retrieval metrics, particularly Recall and Normalized Disjunctive Cumulative Gain, to better measure the impact of the tool in service registries.

## Acknowledgements

We acknowledge the financial support by ANCPyT through grant PICT-2012-0045.

## Referencias

[1] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: A research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 2, pp. 223–255, 2008.

[2] C. Wu, "Wsdl term tokenization methods for ir-style web services discovery," *Science of Computer Programming*, vol. 77, no. 3, pp. 355 – 374, 2012.

[3] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising wsdl documents: Why and how," *IEEE Internet Computing*, vol. 14, pp. 48–56, 2010.

[4] C. Mateos, M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Measuring the impact of the approach to migration in the quality of web service interfaces," *Enterprise Information Systems*, vol. In Press, 2012.

[5] J. M. Rodriguez, M. Crasso, and A. Zunino, "An approach for web service discoverability anti-patterns detection," *Journal of Web Engineering*, vol. 12, no. 1–2, pp. 131–158, 2013.

[6] C. Mateos, M. Crasso, A. Zunino, and J. L. Ordiales Coscia, "Revising wsdl documents: Why and how - part ii," *IEEE Internet Computing*, vol. 17, no. 5, pp. 46–53, 2013.

[7] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Improving web service descriptions for effective service discovery," *Science of Computer Programming*, vol. 75, no. 11, pp. 1001 – 1021, 2010.

[8] J. M. Rodriguez, M. Crasso, C. Mateos, A. Zunino, M. Campo, and G. Salvatierra, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2013, ch. The SOA Frontier: Experiences with Three Migration Approaches, pp. 126–152.

[9] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best practices for describing, consuming, and discovering web services: A comprehensive toolset," *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.

[10] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751 –761, Oct. 1996.

[11] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, 2003.

[12] G. A. Miller, "Wordnet: A lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[13] H. S. Chae, Y. R. Kwon, and D. H. Bae, "A cohesion measure for object-oriented classes," *Software: Practice and Experience*, vol. 30, no. 12, pp. 1405–1431, 2000.

[14] E. E. Allen and R. Cartwright, "Safe instantiation in generic java," *Science of Computer Programming*, vol. 59, no. 1-2, pp. 26–37, 2006.

[15] M. Crasso, A. Zunino, and M. Campo, "Combining query-by-example and query expansion for simplifying Web Service discovery," *Information Systems Frontiers*, vol. 13, no. 3, pp. 407–428, 2011.

[16] E. Agichtein, E. Brill, S. Dumais, and 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 (SIGIR '06)*. ACM Press, 2006, pp. 3–10.