The EasySOC Project: A Rich Catalog of Best Practices for Developing Web Service Applications*

Juan Manuel Rodriguez, Marco Crasso, Cristian Mateos, Alejandro Zunino, Marcelo Campo
ISISTAN Research Institute - UNICEN University,
Tandil, Buenos Aires, Argentina
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
e-mails [jmrodri|mcrasso|cmateos|azunino]@conicet.gov.ar
mcampo@exa.unicen.edu.ar

Abstract

The Service-Oriented Computing (SOC) paradigm has gained a lot of attention in the software industry since SOC represents a novel way of architecting distributed applications. SOC is mostly materialized via Web Services, which allow developers to expose their application as building blocks for other applications by defining a clear and public interface. Although conceptually and technologically mature, SOC still lacks adequate development support from a methodological point of view. We present the EasySOC project: a catalog of guidelines to build service-oriented applications and services. This catalog synthesizes best SOC development practices that arise as a result of several years of research in fundamental SOC-related topics, namely WSDL-based technical specification, Web Service discovery and Web Service outsourcing. In addition, we describe a plug-in for the Eclipse IDE that has been implemented to simplify the utilization of the guidelines. We believe that the practical nature of the guidelines, the empirical evidence that supports them, and the availability of IDE-integrated tools that enforces them will help software practitioners to rapidly exploit our ideas for building real SOC applications.

Keywords: SERVICE-ORIENTED COMPUTING, SERVICE-ORIENTED DEVELOPMENT GUIDELINES, WEB SERVICES, WSDL ANTI-PATTERNS, WEB SERVICE DISCOVERY, WEB SERVICE CONSUMPTION

1 Introduction

Service-Oriented Computing (SOC) [1] is a relatively new computing paradigm that has radically changed the way applications are architected, designed and implemented. SOC has mainly evolved from component-based software engineering by introducing a new kind of building block called *service*, which represents functionality that is described, discovered and remotely consumed by using standard protocols. Service-oriented software systems started as a more flexible and cost-effective alternative for developing Web-based applications, but their usage eventually spread to gave birth to a wave of contemporary infrastructures and notions including Service-Oriented Grids and Software-As-A-Service [2].

The common technological choice for materializing the SOC paradigm is Web Services, i.e. programs with well-defined interfaces that can be published, discovered and consumed by means of ubiquitous Web protocols [1] (e.g. Simple Object Access Protocol (SOAP) [3]). The canonical model underpinning Web Services encompasses three basic elements: service providers, service consumers and service registries (see Figure 1). A service provider, such as a business or an organization, provides meta-data for each service, including a description of its technical contract in Web Service Description Language (WSDL) [4]. WSDL is an XML-based language that allows providers to describe the functionality of their services as a set of abstract operations with inputs and outputs, and to specify the associated binding information so that consumers can actually consume the offered operations.

To make their WSDL documents publicly available, providers employ a specification for service registries, called Universal Description, Discovery and Integration (UDDI), whose central purpose is the representation of meta-data about Web Services. Apart from a meta-data model, UDDI defines an inquiry Application Programming Interface (API), in terms of the WSDL, for discovering services. Consumers use the inquiry API to find services that match their functional needs, select one, and then consume its operations

*This paper is an extended version of the paper presented at the XXIX International Conference of the Chilean Computer Science Society - SCCC/JCC'10

by interpreting the corresponding WSDL description. Both the model and the API are built on Web Service technologies, as the aim of the WSDL and UDDI is to offer standards to enable interoperability among applications and services across the Web. As a consequence, for example, an application implemented in a programming language can talk to a Web Service developed in another language. Ideally, such interoperability levels would allow consumers to switch among different providers of the same functionality, according to non functional requirements such as cost per service consumption, response time or availability, without modifying the applications involved.

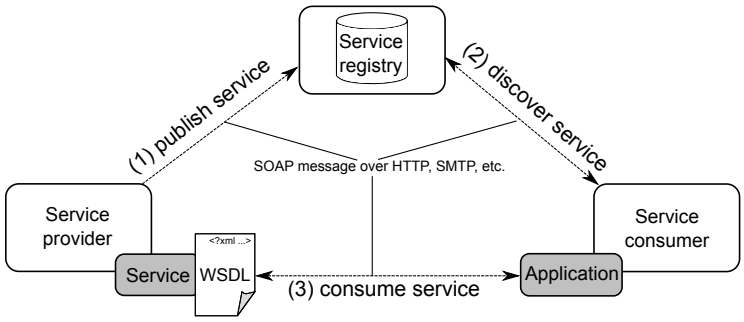


Figure 1: The Web Services model.

Unfortunately, the promises of Web Services of guaranteeing loose coupling among applications and services, providing agility to respond to changes in requirements, offering transparent distributed computing and lowering ongoing investments are still eclipsed by the high costs of outsourcing Web Services of current approaches for service-enabling applications as well as the ineffectiveness of Web Service publication systems. On one hand, unless appropriately specified by providers, service meta-data can be counterproductive because a low-quality WSDL document tends to obscure the purpose of a service, thus hindering its adoption. For example, a WSDL document without much comment of its operations can make the associated Web Service difficult to be discovered and understood. On the other hand, service consumers often have to invest much effort into providing code to invoke discovered Web Services afterward. Moreover, the outcome of the second task is software containing service-aware code. Therefore, the software is more difficult to test and to modify during its maintenance phase especially if the service consumer wants to replace a service for another functional-equivalent service.

In this paper, we describe EasySOC that is a set of provider and consumer guidelines for avoiding these problems. Roughly, these guidelines represent a compilation of best practices for simplifying the activities illustrated in the arcs of Figure 1, while improving the quality of the artifacts implementing services and consumers' applications. EasySOC is based on our previous research in WSDL-based technical contract design and specification [5], Web Service discovery [6] and service-oriented development and programming [7, 8].

Complementary, the contribution of this paper is to provide a uniform, conceptualized and synthesized view of these findings to provide clear and precise hints of how to adequately exploit the SOC paradigm and its related technologies regardless their usage context, i.e. when implementing services or applications. At the same time, another contribution is a concrete materialization of these hints into a software tool so as to enforce the promoted best practices. With respect to the latter, we have built a plug-in for the popular Eclipse IDE and the Java language, thus we believe our ideas can be readily employed in the software industry [9]. The software can be downloaded from `http://sites.google.com/site/easysoc`.

The rest of the paper is structured as follows. The next section focuses on discussing the aforementioned guidelines, emphasizing on clarifying their scope and the development scenarios in which they are applicable. Later, Section 3 presents the EasySOC Eclipse plug-in and its modules. Then, Section 4 surveys relevant related efforts. Finally, Section 5 concludes the paper.

2 The EasySOC project

Even when Web Service technologies are far more mature and reliable than they were years ago, the definition of guidelines for developing service-oriented software is still an incipient research topic. Thus, the following paragraphs present a catalog of identified best practices for SOC development, which are related to the roles and activities that are commonly performed by developers of both services and consumers' applications. Schematically, according to the model of Figure 1, two distinctive roles are established: providers and consumers. Providers are responsible for making a piece of software publicly available as a Web Service, while ensuring that such a service can be discovered and understood by third-parties. Consumers are responsible for discovering and incorporating external services into their applications, or from now on *client applications*. Sometimes the same actors can play both roles, as occurs when developing services that need of other services to accomplish the functionality they expose.

Depending on the role(s) played by a SOC developer, there are three possible different development scenarios. Table 1 lists which of the EasySOC guidelines should be followed in each scenario.

Table 1: SOC usage scenarios and the EasySOC guidelines.

Scenario/Guidelines	**Guidelines for service publication**	**Guidelines for service discovery**	**Guidelines for service consumption**
Developer only exposes a functionality as a service	Yes	No	No
Developer only consumes services	No	Yes	Yes
Developer exposes as a service a functionality that consumes other services	Yes	Yes	Yes

2.1 Guidelines for service publication

Broadly, service publication is an activity that comes after a particular service has been developed, or in other words, the interface and the implementation of the service have been built. Methodologically, and depending on the order in which interfaces and implementations are derived, services can be constructed either based on *contract-first* or *code-first*. Contract-first is a method that encourages publishers to first derive the WSDL contract of a service and then supply an implementation for it. Alternatively, with code-first, a publisher first implements a Web Service and then generates the corresponding service contract by automatically extracting and deriving the interface from the implemented code. This means that WSDL documents are not directly created by humans but are instead automatically generated via programming language-dependent tools.

Regardless of the approach employed to build services, bad practices might manifest themselves in the resulting WSDL documents. With contract-first Web Services, this usually arises as a consequence of poorly described WSDL documents supplied by developers [5]. Moreover, with code-first Web Services, such practices result from poor implementation decisions or deficient WSDL generation tools. Precisely, Sections 2.1.1 and 2.1.2 explain the EasySOC guidelines for avoiding these bad practices when dealing with contract-first or code-first services, respectively.

2.1.1 Guidelines for contract-first WSDL construction

Many of the problems related to the efficiency of standard-compliant approaches to service discovery stem from the fact that the WSDL specification is incorrectly or partially exploited by providers. The WSDL specification allows providers to describe the service functionality as a set of *port-types*. A port-type arranges different *operations* whose invocation is based on exchanging *messages*: one message with input data, other with the result, and another with error information, optionally. Port-types, operations and messages must be named with unique names. Messages consist of *parts* that are arranged according to specific data-types defined using the XML Schema Definition (XSD) language. XSD offers constructors for defining simple types (e.g., integer and string), and more elaborate mechanisms for defining complex elements. Data-type definitions can be put into a WSDL document or into a separate file and imported from any WSDL document afterward. The grammar of the WSDL[1] can be summarized as follows:

```
<documentation .... />?
<types>?
    <documentation .... />?
    <schema .... />*
</types>
<message name="nmtoken">*
    <documentation .... />?
    <part name="nmtoken" element="qname"? type="qname"?/>*
</message>
<portType name="nmtoken">*
    <documentation .... />?
    <operation name="nmtoken">*
        <documentation .... />?
        <input name="nmtoken"? message="qname">?
            <documentation .... />?
        </input>
        <output name="nmtoken"? message="qname">?
            <documentation .... />?
        </output>
        <fault name="nmtoken" message="qname">*
            <documentation .... />?
        </fault>
    </operation>
</portType>
```

Although the intuitive importance of properly describing services, some practices that attempt against services discoverability and understandability, such as poorly commenting offered operations or using unintelligible naming conventions, are frequently found in publicly available WSDL documents. No silver bullet can guarantee that potential consumers of a Web Service will effectively

[1]Note that "?" means optional and "*" means none or many.

discover, understand and access it. However, we have empirically shown that a WSDL document can be improved to simultaneously address these issues by following six steps [10]:

1. Separating the schema –i.e. XSD code– from the definitions of the offered operations.
2. Removing repeated WSDL and XSD code.
3. Putting error information within Fault messages and only conveying operation results within Output ones.
4. Replacing WSDL element names with self-explanatory names if they are cryptic.
5. Moving non-cohesive operations from their port-types to a new port-type.
6. Properly commenting the operations.

The first step means moving complex data-type definitions into a separate XSD document, and adding the corresponding import sentence into the WSDL document. However, when data-types are not going to be reused or are very simple, they can be part of the WSDL document to make it self-contained.

The second step deals with redundant code in both the WSDL document and the schema. Repeated WSDL code usually stems from port-types tied to a specific invocation protocol, whereas redundant XSD is commonly a result of data definitions bounded to a particular operation. Therefore, repeated WSDL code can be removed by defining a protocol-independent port-type. Similarly, to eliminate redundant XSD code, repeated data-types should be abstracted into a single one. Both changes require updating the references in the WSDL document to the new defined elements that replace the redundant elements.

The third step is to separate error information from output information or service invocation results. To do this, error information should be removed from Output messages and placed on Fault ones, a special construct provided by WSDL to specify errors and exceptions. Moreover, as many Fault messages as kinds of errors exist should be defined for the operations of the Web Service.

The fourth step aims to improve the representativeness of WSDL element names by renaming non-explanatory ones. Grammatically, the name of an operation should be in the form <verb> "+" <noun>, because an operation is essentially an action. Furthermore, message, message part and data-type names should be a noun or a noun phrase because they represent the objects on which the operation executes. Additionally, the names should be written according to common notations, and their length should be between 3-15 characters because this facilitates both automatic analysis and human reading, respectively. With respect to the former hint, the name "theelementname" should be rewritten for example as "theElementName" (camel casing).

The fifth step is to place operations in different port-types based in their cohesion. To do this, the original port-type should be divided into smaller and more cohesive port-types. This step should be repeated while the new port-types are not cohesive enough.

Finally, all operations must be well commented. An operation is said to be well commented when it has a concise and explanatory comment, which describes the semantics of the offered functionality. Moreover, as WSDL allows developers to comment each part of a service description separately, then a very good practice is to place every <documentation> tag in the most restrictive ambit. For instance, if the comment refers to a specific operation, it should be placed in that operation.

It is worth noting that, except for steps 4 and 6, the other steps might require to modify service implementations. Moreover, as a result of applying these guidelines, there will be two versions of a revised service description. Despite being out of the scope of this paper, some version support technique is necessary to allow service consumers that use the old service version to continue using the service until they migrate to the new version.

2.1.2 Guidelines for code-first WSDL generation

As explained in the previous Section, by having control of the WSDL document representing a service, its provider can use six specific steps to improve the discoverability and understandability of the published service. On the other hand, when building code-first services, such control is partial because WSDL documents are automatically derived from (manually-implemented) service codes by using generation tools. Formally, a typical code-first tool performs a mapping T :

$$T : C \rightarrow W, \tag{1}$$

Mapping T from $C = \{M(I_0,R_0),..,M_N(I_N,R_N)\}$ or the main module implementing a service generates a WSDL document $W = \{O_0(I_0,R_0),..,O_N(I_N,R_N)\}$ or the software artifact describing the service. Furthermore, W contains a *port-type* for the service implementation class C, having as many *operations* O as public methods M are defined in the class. Moreover, each *operation* of W will be associated with one input *message* I and another return *message* R, while each *message* conveys an XSD type that stands for the parameters of the corresponding class method. Examples of code-first tools are WSDL.exe, Axis' Java2WSDL and gSOAP [11], which generate WSDL documents from C#, Java and C++ source code, respectively. Naturally, each tool implements T in a particular manner mostly because of the different characteristics of the involved programming languages.

Although in this context generation tools control WSDL specification, we found that providers can indirectly avoid or reduce the impact of the abovementioned WSDL-level bad practices by following certain programming guidelines at service implementation time. It is worth noting that the guidelines are essentially aimed at Java, which is the language at which our research is currently scoped. Below we list these guidelines:

1. Replacing main class and method parameter names with self-explanatory names if they are cryptic.
2. Moving non-cohesive methods from their classes to a new class.
3. Properly commenting the purpose of main classes, methods and method parameters.
4. Replace data-types declared as Object with a concrete data-type whenever possible.

Similarly to the case of the guidelines for publishing contract-first services, the first step indirectly helps in improving the representativeness of WSDL element names, which are derived from main classes and method parameters. To this end, the same actions should be taken, i.e. revising the grammar of such identifiers.

The second step, on the other hand, allows developers to improve the functional cohesion of the resulting WSDL descriptions by first checking the functional cohesion of their implementation classes. Note that refactoring a class for better cohesion might imply splitting it into several classes, which in turn depending on the generation tool being used either leads to a single WSDL document with several port-types or several WSDL documents each having a single port-type.

Moreover, the third step encourages developers to provide descriptive comments for service implementations. Like the WSDL standard, which allows developers to comment the different parts of the interface of a contract-first service, Java provides the Javadoc tool, a standard language facility that includes tags for commenting program elements.

Finally, the fourth step means simply to specify data-types (mostly method parameters) in a precise way. Developers must avoid the use of Object and instead rely on concrete data-types. In addition, data-types such as Vector, List, Hashtable, etc., should be replaced with their generic-aware counterparts, i.e. Vector<X>, List<Y>, Hashtable<K,V>. Likewise, the generics must be instantiated with the most restrictive class/interface as well, i.e. using Vector<Object> will produce the same negative effect in the WSDL document as simply using Vector.

2.2 Guidelines for service discovery

Queries play an important part in the process of service discovery since service consumers greatly benefit from generating clear and explanatory descriptions of their needs. This is because the underpinnings of UDDI-based registries rely upon the descriptiveness of the keywords conveyed in both publicly available service interfaces and queries.

We have empirically proved that the source code artifacts of client applications usually carry relevant information about the functional descriptions of the potential services that can be discovered and, in turn, consumed from within applications [6]. The idea is that developers should be focused on building the logic of their applications, while using automatic heuristics to pull out keywords standing for queries of the required services from the code implementing such applications. In this line, best practices for building an application that contains useful information about the services the application needs comprise [6]:

1. Defining the expected interface of every application component that is planned not to be implemented, but outsourced to a Web Service.
2. Revising the functional cohesion between the implemented (i.e. internal) components that directly invoke, and hence depend on the interfaces of, the components defined in step 1.
3. Naming and commenting each defined interface and internal component by using self-explanatory names and comments, respectively.

The first step encourages developers to think of a third-party service as any other regular component providing a clear interface to its operations. The idea of defining a functional interface before knowing the actual exposed interface of a service that fulfills an expected functionality aligns with the *Query-by-Example* approach to create queries. This approach allows a discoverer to search for an entire piece of information based on an example in the form of a selected part of that information. This concept suggests that because of the structure inherent to client applications and Web Service descriptions in WSDL, the expected interface can be seen as an example of what a consumer is looking for. This is built on the fact that, via WSDL, publishers can describe their services as object-oriented interfaces with methods and arguments. Therefore, in the context of client applications, the defined interfaces stand for *examples*.

The second step bases on an approach for automatically augmenting the quantity of relevant information within queries called *Query Expansion*, which relies on the expansion of extracted examples by gathering information from the source code representing internal components that directly interact with the interfaces representing external services. The reasoning that supports this

mechanism is that expanding queries based upon components with strongly-related and highly-cohesive operations should not only preserve, but also improve, the meaning of the original query. Therefore, the second step deals with ensuring that defined interfaces are strongly-related and highly-cohesive with those components that depend on them.

The above two steps deal with identifying the source code parts of an application that are likely to contain relevant information for generating queries and discovering Web Services afterward. Also, a third step exists for checking that the identified code parts actually have relevant information for that purpose. Since automatic query generation heuristics gather keywords from operation names and comments present in source codes, developers should follow conventional best practices for naming and commenting their code.

2.3 Guidelines for service consumption

Maintaining client applications can be a cumbersome task when they are tied to specific providers and WSDL documents. The common approach to call a Web Service from within an application is by interpreting its associated WSDL document with the help of invocation frameworks such as WSIF, CXF [12], or the .NET Web Services Description Language Tool (WSDL.exe)[2]. These frameworks succeed in hiding the details for invoking services, but they still fail at isolating internal components from the interfaces of the services. Consequently, applications result in a mix of pure logic and sentences for consuming Web Services that depend on their operation signatures and data-types. This approach leads to client applications that are subordinated to third-party service interfaces and must be modified and/or re-tested every time a provider introduces changes. In addition, this also hinders service replaceability, which means how easily a service could be replaced with another functionality equivalent service.

In this sense, we have shown that the maintenance of service-oriented applications can be facilitated by following certain programming practices when outsourcing services [8]:

1. Defining the expected interface of every component that is planned to be outsourced.
2. Adapting the actual interface of a selected service to the interface that was originally expected, i.e. the one defined in the previous step.
3. Seamlessly injecting adaptation code into each internal component that depends on the expected interface.

Step 1 provides a mean of shielding the internal components of an application from details related to invoking third-party services. To do this, a functionality that is planned to be implemented by a third-party service should be programmatically described as an abstract interface. Note that this is the same requirement as the first step of Section 2.2. Accordingly, internal application components depending on such an abstractly-described functionality consume the methods exposed by its associated interface, while adhering to operation names and input/out data-types declared in it.

The second step takes place after a service has been selected. During this step, developers should provide the logic to transform the operation signatures of the actual interface of the selected service to expected the interface defined previously. For instance, if a service operation returns a list of integers, but the interface defined at step 1 returns an array of floats, the developer should code a service adapter that performs the type conversion. By properly accomplishing steps 1 and 2, client components depend on neither specific service implementations nor interfaces. Therefore, from the perspective of the application logic, services that provide equivalent functionality can be transparently interchangeable at the expense of building specific adapters.

Finally, the third step is for separating the functional code of an application from configuration aspects related to binding a client component that depends on an interface with the adapter component in charge of adapting it into a selected service. A suitable form of doing this, in terms of source code quality, involves delegating to a software layer or container the administrative task of assembling interfaces, internal components and services together.

In the following section we describe a software tool, implemented as a plug-in for the Eclipse IDE, which enforces the aforementioned guidelines for developing SOC applications and Web Services written in Java.

3 The EasySOC Eclipse plug-in

The EasySOC Eclipse plug-in comprises four modules, each one associated to the set of guidelines explained before. Sections 3.1 through 3.4 discuss the design and implementation of these modules. Although EasySOC guidelines are valid for any Web Service capable platform, it is important to notice that the EasySOC Eclipse plug-in is a software tool aimed only to the Java platform. Hence, some of the implementation details are platform dependant. However, it would be possible to implement these tools for other development platforms, such as .Net.

[2].NET WSDL.exe, `http://msdn.microsoft.com/en-us/library/7h3ystb6(v=VS.71).aspx`

3.1 The WSDL discoverability Anti-Patterns' Detector

The Anti-patterns' Detector is the EasySOC module for automatically checking whether the WSDL document depicting a Web Service technical contract conforms to the guidelines of Section 2.1.1. The module bases on the fact that the goal of Section 2.1.1 guidelines is to avoid the occurrence of the WSDL document discoverability anti-patterns introduced in [10] within contract-first service descriptions. Besides measuring the impact of each anti-pattern on service discovery, this study [10] assessed the implications of anti-patterns on developers' ability to make sense of WSDL documents. The catalog consists of eight anti-patterns and provides a name, a problem description, and a sound refactoring procedure for each anti-pattern. Although the results of the study motivate anti-patterns refactoring, manually looking for an anti-pattern in WSDL documents might be a time consuming and complex task. Thus, the Anti-patterns' Detector comprises heuristics to automatically detect the anti-patterns in the aforementioned catalog.

Since these heuristics are based on the anti-pattern definition, they can be classified according to the analysis required to detect the anti-patterns. In this sense, a taxonomy comprising types of anti-patterns was derived [10]. Basically, anti-patterns can be divided into two categories: those that can be detected by analyzing only the structure of a WSDL document, and anti-patterns whose detection requires a semantic analysis of the names and comments in the WSDL document.

The heuristics to detect the first kind of anti-patterns are simple rules based on the commonest anti-pattern occurrence form. The problems related to these anti-patterns are redundant XML code for defining both data-types and port-types, data-types embedded in a WSDL document, non-commented operations, and data-types that allow transferring data of any type (*Whatever types* in EasySOC terminology).

Firstly, the rule that detects redundant port-types verifies that any pair of port-types has the same number of operations and that they are equally named. In this case, the heuristic does not verify the similarity between the messages of the port-types because they are likely to change in accord with the underlying binding protocol. Furthermore, the problem of embedded data-types is detected by checking that no external XSD document(s) are referenced from within a WSDL document. To detect lacking comments, the associated heuristic simply verifies that all the operations have a non-empty `<documentation>` tag. Finally, detecting the *Whatever types* anti-pattern involves detecting its two forms. One form is when a data type tag is defined with the primitive type "anyType" indicating that any type is a valid contain for the typed tag. The other form of this anti-pattern is when a data-type definition includes an `<any>` tag, which means that any valid XML is valid at that point. Therefore, if an `<any>` tag is present or some tag have "anyType" as a value of its "type" property, this anti-pattern is said to be present in the WSDL document.

As it was previously mentioned, detecting the remaining anti-patterns requires analyzing the semantics of names and comments. Basically, there are three problems that are detected by the associated heuristics: two naming issues, operations of different domains in the same port-type, and fault information within standard output messages. Firstly, our heuristic deals with names being too short or too long, using a rule to check that each name has a length between 3-30 characters is provided. In addition, our heuristic concerns name structure, i.e. message part names should be nouns or noun phrases, while operations should be named with a verb plus a noun. This is verified by using a probabilistic context free grammar parser [13]. For example, Figure 2 depicts the parsing trees of different message part names generated by the parser. The first and second names do not present problems, whereas the third name does because it starts with a verb.

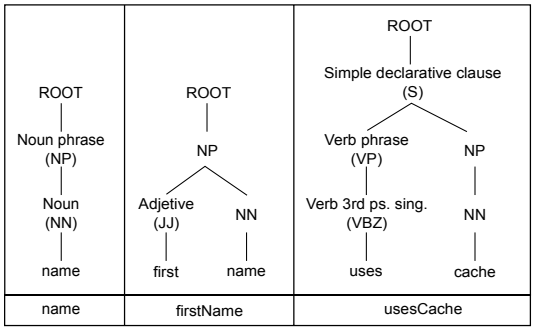


Figure 2: Parsing trees of message part names.

Secondly, our heuristic for determining whether two operations belong to the same domain is based on a text classification technique, because the only "semantic" information about an operation are its names (operation name and message name) and comments. In particular, the Rocchio's TF-IDF classifier has been selected because empirical studies have shown that it outperforms other classifiers in the Web Services area [8]. Rocchio's TF-IDF represents textual information as vectors, in which each dimension stands for a term and its magnitude is the weight of the term related with the text. Having represented all the textual information of a domain as vectors, the average vector, called centroid, is built for representing the domain. Then, the domain of an operation is deduced by representing it as a vector and comparing it to each domain centroid. Lastly, the domain associated with the most similar average vector is returned as the domain of that operation.

Finally, our heuristic for detecting error information within output messages checks whether an operation has no fault message defined, and the comments or some name related with the output contains one of the following words, which are commonly related with error conditions while executing a service: *error, fault, fail, exception, overflow, mistake* and *misplay*.

These heuristics have been experimentally validated with two real-world data-sets. The first validation was intended to empirically prove that the heuristics can effectively detect the anti-patterns. Therefore, we used the original WSDL data-set employed to assess the impact of service descriptions containing anti-patterns on service discovery [10]. The second experiment was designed to determine (if any) what is the correlation between our anti-patterns and the metrics proposed in [14], which were obtained using two benchmarking tools, namely ParaSoft SOATest and WS-I Interoperability Testing Tools; to perform this experiment, we used the WSDL document data-set presented in [15].

The first validation showed that the heuristics have an accuracy of 98.5%, on average. The methodology followed in the evaluation involved manually analyzing each WSDL document to identify the anti-patterns it has, peer-reviewing manual results afterward (at least three different people reviewed each WSDL document), automatically analyzing WSDL documents based on the proposed heuristics, and finally comparing both manual and automatic results. Achieved results are shown in Table 2 by using a confusion matrix. Each row of the matrix represents the number of WSDL documents that were automatically classified using the heuristic associated with a particular anti-pattern. In addition, the columns of the matrix show the results obtained manually, i.e. the number of WSDL documents that actually had each anti-pattern. Results are organized per anti-pattern, and if a WSDL document has an anti-pattern it is classified as "Positive", otherwise it is classified as "Negative". When the manual classification for a WSDL document is equal to the automatic one, it means that the heuristic accurately operates for that WSDL document.

Table 2: Anti-pattern detection: Confusion matrixes.

Automatic detection results per anti-pattern		**Manual detection results**	
		Negative	Positive
Enclosed data model	Negative	116	6
	Positive	0	270
Redundant port-types	Negative	161	4
	Positive	0	227
Redundant data models	Negative	221	2
	Positive	3	166
Whatever types	Negative	339	0
	Positive	3	50
Lack of comments	Negative	135	0
	Positive	0	257
Low cohesive operations in the same port-type	Negative	272	10
	Positive	78	32
Ambiguous names	Negative	67	0
	Positive	9	316
Undercover fault information within standard messages	Negative	351	3
	Positive	4	34

In this experiment, we used a data-set of 392 WSDL documents [10]. Once each heuristic was applied on this data-set, we built the confusion matrixes. Then, we assessed the accuracy and false positive/negative rates for each matrix. The accuracy of each heuristic was computed as the number of classifications matching over the total of analyzed WSDL documents. For instance, the accuracy of the *Redundant data model* heuristic was $\frac{221+166}{221+2+3+166} = 0.987$. The heuristic for detecting *Low cohesive operations within the same port-type* anti-pattern achieved the lowest accuracy: 0.775. One hypothesis that could explain this value relates to potential errors introduced by the classifier, thus more experiments are being conducted. Nevertheless, as mentioned before, the average accuracy for all the heuristics was 0.958.

The false positive rate is the proportion of WSDL documents that an heuristic has wrongly labeled as having the corresponding anti-pattern. In opposition, the false negative rate is the percentage of WSDL documents that an heuristic has wrongly labeled as not having the corresponding anti-pattern. A false negative rate equals to 1.0 means that a detection heuristic has missed all anti-pattern

occurrences. Therefore, the lower the achieved values the better the detection effectiveness. The average false positive rate was 0.036, and the average false negative rate was 0.052, which we believe are encouraging.

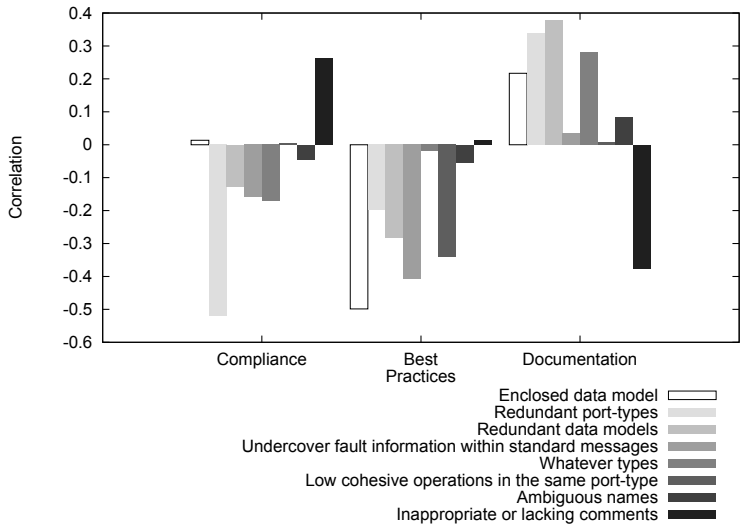


Figure 3: Correlation among Anti-patterns and the Al-Masri and Mahmoud's quality metrics.

In the second validation, we have calculated the correlation between anti-pattern occurrences and the Al-Masri and Mahmoud's quality metrics [14] and data-set [15]. This data-set consists of 1822 Web Services that exist on the Web, and provides their associated WSDL documents and collected quality metrics. The method used to calculate this correlation was the Pearson product-moment correlation coefficient because we expected to find a direct relationship between Al-Masri and Mahmoud's quality metrics and the anti-patterns. The results showed that some of the Al-Masri and Mahmoud's quality metrics have a significant correlation with the anti-patterns, but others have not. However, when we analyzed these results, the metrics that had no correlation were response time, availability, throughput, successability, reliability, latency, WsRF and service classification (a discrete value from 1 to 4 representing service offering qualities). Since these are technical metrics unrelated to WSDL document quality, it is reasonable that they have no correlation with the anti-patterns. Furthermore, the metrics related to WSDL document quality proposed in [14] are:

- Compliance: The degree to which a WSDL document grammatically conforms to the WSDL specification.
- Best Practices: The degree to which a Web service complies with WS-I profile guidelines.
- Documentation: The amount of textual documentation in description tags including service, ports, and operations.

Figure 3 shows the correlation between anti-pattern occurrences and measures taken according to the Al-Masri and Mahmoud's quality metrics. In Al-Masri and Mahmoud's quality metrics, low values stand for low quality, and high values stand for high quality. While, the anti-patterns' variable are zero for not present and one for present in the WSDL document. A correlation value higher than zero means that when one variable rises, the other variable value tends also to rise. While, a correlation value lower than zero means that when one variable rises, the other variable value tends to decrease. It is important to notice that correlation values are neither too high nor too low because we are correlating a discrete-valued variable (anti-pattern occurrences) that cannot have a linear relation to a continue value. Finally, a correlation value near zero means that the values of the variables are independent, i.e., anti-pattern occurrences and Al-Masri and Mahmoud's quality metrics are not related.

When the Compliance metric value is high, WSDL documents tend not to be affected by most of the anti-patterns. The exception of this is the *Enclosed data-model anti-pattern*, for which correlation is near zero, and the *Inappropriate or lacking comments* anti-pattern, which is a highly correlated anti-pattern. The first exception is sound because both options, having enclosed data-model or importing them from an XSD file, are WSDL compliant. While, the problems with documentation might be related to the fact that when documentation is added to a WSDL document the developer should manually modify it, which is an error-prone task.

The Best Practices metric is similar to the Compliance metric because most of the anti-patterns are negatively correlated with it. The only exceptions are *Whatever types* and *Inappropriate or lacking comments* anti-patterns. In this case both correlations are near zero, meaning that this metric and these anti-patterns occurrences are statistically independent. This is probably because Best Practices metric is related to WS-I guidelines that intent to improve the technical interoperability, but not the usability of a Web Service; and these anti-patterns are precisely connected to WSDL document readability and understandability.

The Documentation metric, which represents the percentage of elements in a WSDL document that contain comments, is only correlated negatively to the *Inappropriate or lacking comments* anti-pattern. This is expected because if a developer introduces

comments to a WSDL document, they are intended to be read. The other anti-patterns are either not correlated or have a positive correlation. The positive correlation might stem from that the developer must manually modify the WSDL document to improve the comments, and modifying a WSDL document is an error-prone task.

3.2 The Service Implementation Watcher

The EasySOC Watcher module is intended for checking whether code-first service implementations comply with the guidelines of Section 2.1.2. This module enforce good coding practices that results in better generated WSDL documents. The code quality is measured using well-known Object-Oriented (OO) class-level metrics, such as the Chidamber and Kemerer's metric catalog [16] and some of our own, on a service code *C*, for allowing a provider to have an estimation of how the resulting WSDL document *W* will be like in terms of WSDL discoverability anti-pattern occurrences. The reader should recall that WSDL generation tools rely on a mapping *T* that relates *C* with *W*. The approach to perform this estimation is based on a statistical analysis that shows a significant correlation among the Weighted Methods Per Class (WMC), Coupling Between Objects (CBO), and Abstract Type Count (ATC) metrics and some anti-patterns. This correlation was experimentally confirmed by using a real-world Web Service data-set.

The WMC [16] metric counts the methods of a class. The experiments have empirically confirmed that a greater number of methods within a service implementation main class increases the probability that any pair of them are unrelated, i.e. having weak cohesion. Since *T*-based code-first tools map each method onto an operation, a higher WMC might increase the possibility that resulting WSDL documents have low cohesive operations.

CBO [16] counts how many methods or instance variables defined by other classes are accessed by a given class. Code-first tools based on *T* include in resulting WSDL documents as many XSD definitions as objects are exchanged by service classes' methods. The experiments have empirically shown that increasing the number of external objects that are accessed by service classes might increase the likelihood of data-type definitions within WSDL documents.

Finally, ATC is a metric of our own that computes the number of method parameters that do not use concrete data-types, or use Java generics with type variables instantiated with non-concrete data-types. We have defined the ATC metric after noting that some *T*-based code-first tools map abstract data-types and badly defined generics onto xsd:any constructors, which have been identified as root causes for the *Whatever types* anti-pattern [10, 17].

Returning to the experiment settings, the data-set consists of around 90 different real services whose implementation was collected via two code search engines, namely the Merobase component finder (`http://merobase.com`) and the Exemplar engine [18]. Merobase allows users to harvest software components from a wide variety of sources (e.g. Apache, SourceForge, and Java.net) and has the unique feature of supporting interface-driven searches, i.e. searches based on the abstract interface that a component should offer, apart from that of based on the text in its source code. On the other hand, Exemplar relies on a hybrid approach to keyword-based search that combines the benefits of textual processing and intrinsic qualities of code to mine repositories and consequently returns complete projects. Complementary, we collected projects from Google Code.

Some of the retrieved projects actually implemented Web Services, whereas other projects contained granular software components such as EJBs, which were "servified" to further enlarge the data-set. After collecting the components and projects, we uniformed the associated services by explicitly providing a Java interface in order to facade their implementations. Each WSDL document was obtained by feeding the Java2WSDL tool with the corresponding interface. All in all, the data-set provided the means to perform a significant evaluation in the sense that the different Web Service implementations came from real-life developers.

Table 3: Correlation among Anti-patterns and the employed OO metrics.

	WMC	CBO	ATC
Ambiguous names	**0.86**	0.42	0.25
Enclosed data model	0.41	**0.98**	0.12
Low cohesive operations in the same port-type	**0.61**	0.38	0.12
Redundant data models	**0.79**	0.33	0.15
Whatever types	0.50	0.35	**0.60**

For the correlation analysis, we calculated the Spearman's rank correlation coefficient to establish the existing relations between the two kind of variables of our model, i.e. the OO metrics (independent variables) and the anti-patterns (dependent variables). The list of anti-pattern occurrence per WSDL document was obtained by using the WSDL discoverability Anti-Patterns' Detector presented in the previous Section. Table 3 depicts the correlation factors among the studied OO metrics. The cell values in bold are those coefficients which are statistically significant at the 5% level, i.e. p-value < 0.05, which is a common choice when performing statistical studies [19]. These correlation factors clearly show that the metrics studied are not statistically independent and, thereby,

capture redundant information. In other words, if a group of variables in a data-set are strongly correlated, these variables are likely to measure the same underlying dimension (i.e. cohesion, complexity, coupling, etc.).

The correlation among the WMC, CBO, and ATC metrics and the anti-patterns, which were found to be statistically significant for the analyzed Web Service data-set, suggests that an increment/decrement of the metric values taken on the code of a Web Service directly affects anti-pattern occurrence in its code-first generated WSDL document. Then, the Watcher module re-calculates these metrics for service implementation classes every time a class changes. Once these metrics are calculated, if metric values increase, the module assumes that the risk of generating a WSDL document having an anti-pattern has increased as well. In this case, the Watcher suggests developers to refactor their service implementation according to the steps listed in Section 2.1.2.

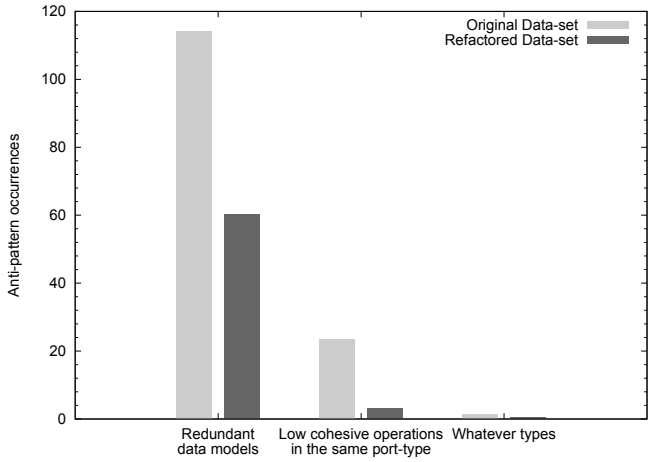


Figure 4: Anti-patterns that were mitigated by refactoring service implementations.

We performed some source code refactorings driven by these OO metrics on our data-set so as to quantify the effect on anti-pattern occurrence. For the sake of representativeness, we modified the services that presented all anti-patterns at the same time, which accounted for a 30% of the entire data-set. However, some code-first tools do not allow avoiding every anti-pattern. For instance, the *Enclosed data model* anti-pattern cannot be removed from code-first WSDL documents when using Java2WSDL or WSDL.exe.

Figure 4 shows the three anti-patterns that were reduced after refactoring. In a first round of refactoring, we focused on reducing WMC by splitting the services having too much operations into two or more services so that on average the metric in the refactored services represented a 70% of the original value. This refactoring, on average, reduced in 47.26% the occurrences of *Low cohesive operations in the same port-type* anti-pattern. At first sight, this refactoring appeared to have a collateral effect: the fall in the occurrences of the *Redundant data models* anti-pattern. However, this improvement was caused by a limitation of the Anti-Pattern Detector that does not count as an anti-pattern occurrence the case when two services define the same data-type. In contrast, if a service has 2 operations both using the defined data-type twice, the Detector counts 2 anti-pattern occurrences. However, after refactoring, if the service is divided in 2 new services with one operation having the same data-type each, the Detector does not count the anti-pattern.

In a second refactoring round, we focused on 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. Basically, the applied refactoring was to replace arguments declared as Object with a concrete data-type whenever possible. In addition, although replacing parameters declared as Vector, List, Hashtable, etc., with their generic-aware counterparts, i.e. Vector<X>, List<Y>, Hashtable<K,V> and so on would in theory be another sound refactoring, we decided to replace the former with array structures due to limitations of the WSDL generation tool used. Overall, by applying these modifications we were able to decrease the number of occurrences of the *Whatever types* anti-pattern. Aside from these benefits for WSDL document quality, this refactor is also recommended for any Java code [20].

3.3 The Query Builder

The EasySOC Query Builder module gathers information to build service queries from the source code of client applications. This module provides a graphical tool that guides consumers through the generation of the queries. When a consumer selects “Find services for...” by clicking on an interface that stands for an external service to be outsourced (see Figure 5 step 1), a wizard dialog starts. The wizard uses the Eclipse JDT Search Engine[3] for automatically discovering the client-side classes that depend on the interface and presents them to the user. Then, the user selects or discard the resulting classes (Figure 5 step 2). Similarly, the wizard

[3]Eclipse Java Development Tools (JDT), `http://www.eclipse.org/jdt`

presents a list of argument classes. This list is automatically built by analyzing the interface to retrieve the class names associated with each argument. If an argument is neither primitive (e.g., int, long, double, etc.) nor provided within a built-in Java library package (e.g., Vector, ArrayList, String, etc.), it is included in the list of argument classes. Finally, those manually selected classes along with the Java interface are used as input for the text-mining process depicted in the center of Figure 5. The module allows users to customize queries and test the retrieval effectiveness when using different classes as input, making query building interactive or semi-automatic. Alternatively, by clicking on the "Finish" button, the wizard selects all target classes on behalf of the consumer, making query expansion fully automatic.

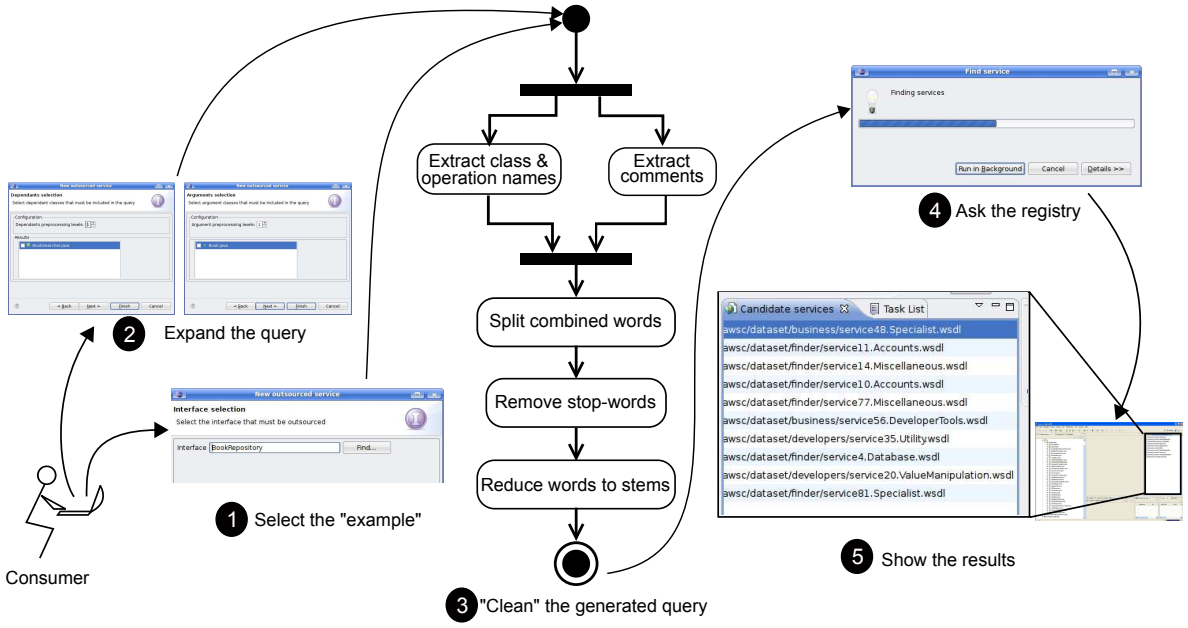


Figure 5: Wizard for generating service queries.

We evaluated the retrieval effectiveness of the Query Builder by using the previous collection of 392 WSDL documents to feed a service registry [6]. Moreover, undergraduate students played the role of service consumers in the context of the "Service-Oriented Computing"[4] course of the Systems Engineering BSc. program at the Faculty of Exact Sciences (Department of Computer Science) of the UNICEN. The students were assigned an exercise consisting on deriving 30 queries, in which each query comprised a Java interface describing the functional capabilities of a potential service. The header and the operations of each interface were commented. For those operations with non-primitive data-types as arguments, their corresponding classes were also commented. Then, for each query, the students implemented and commented the internal components that depended on the interface. This methodology allowed us to separately evaluate five combinations of different sources of terms associated with an *example*, namely its "Interface", "Documentation", "Arguments" and "Dependants". Finally, a fifth alternative was used by combining all these four sources. In this context, documentation does not refer to extra software artifacts but to textual comments embedded within the classes.

To evaluate the discovery performance resulted from employing the different sources of terms, we used the Precision-at-*n*, Recall-at-*n*, *R*-precision and Normalized Recall (NR) information retrieval metrics. In this sense, the goal was to evaluate our Query Builder in terms of the proportion of relevant services in the retrieved list and their positions relative to non-relevant ones. We applied each metric for the 30 queries by individually using each one of the combination of sources (a total of 150 experiments per measure), and then we averaged the results over the 30 queries. As some of these metrics require knowing the set of all services in the collection that are relevant to a given query, we exhaustively analyzed the data-set to determine the relevant services for each query. An important characteristic regarding the evaluation is the definition of "hit", i.e. when a returned WSDL document is actually relevant to the user. We judged a WSDL document as being a hit or not depending on whether its operations fulfilled the expectations previously specified in the Java code. For example, if the consumer required a Web Service for converting from Euros to Dollars, then a retrieved Web Service for converting from Yens to Dollars was not considered relevant, even though these services were strongly related. In this particular case, only Web Services for converting from Euros to Dollars were relevant. Note that this definition of hit makes the validation of our discovery mechanism very strict. Additionally, it is worth noting that for any query there are, at most, 8 relevant services within the data-set. Besides, there are 10 queries that have associated only one relevant service.

Each bar in Figure 6 stands for the averaged metric results that were achieved using a particular query expansion alternative. Achieved results pointed out that by following the conventional Query-By-Example approach to build queries (the alternative named

[4] http://www.exa.unicen.edu.ar/~cmateos/cos

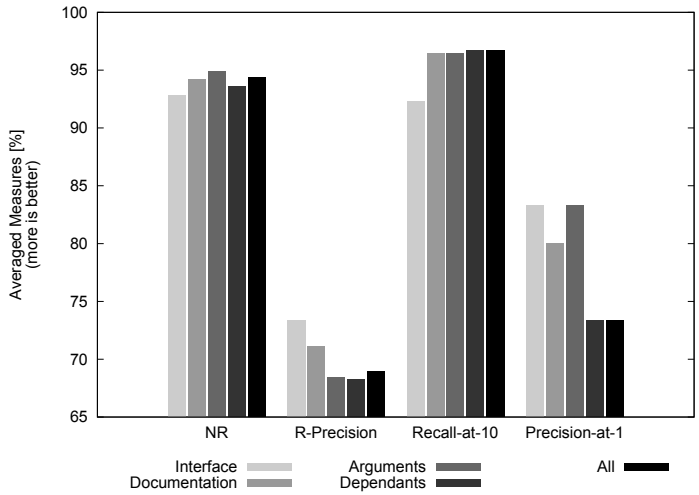


Figure 6: EasySOC Query Builder: Retrieval effectiveness.

"Interface") query-specific results were ranked first. When using more general, elaborated queries via the Query Expansion approach (e.g. the "All" alternative), the chance of including a relevant service at the top of the list decreased as the possibilities of including it before the 11^{th} positions increased. All in all, for this experiment our Query Builder alleviated discovery by concentrating relevant services within a window of 10 candidates.

3.4 The Service Adapter

The Service Adapter module has been created to automatically perform the steps 2 and 3 for the guidelines of Section 2.3. Once a consumer has selected a candidate service, this module performs three different tasks to adapt service interfaces and assemble internal components to it. The first task builds a proxy for the service. Second, the module builds an *adapter* to map the interface of the proxy onto the abstract interface internal components expect. Third, the module indicates a container how to assemble internal components and adapters, which is done through Dependency Injection (DI), a popular pattern for seamlessly wiring software components together that is employed by many development frameworks. Figure 7 summarizes the steps that are needed to proxy, adapt, and inject services into applications or another service implementation.

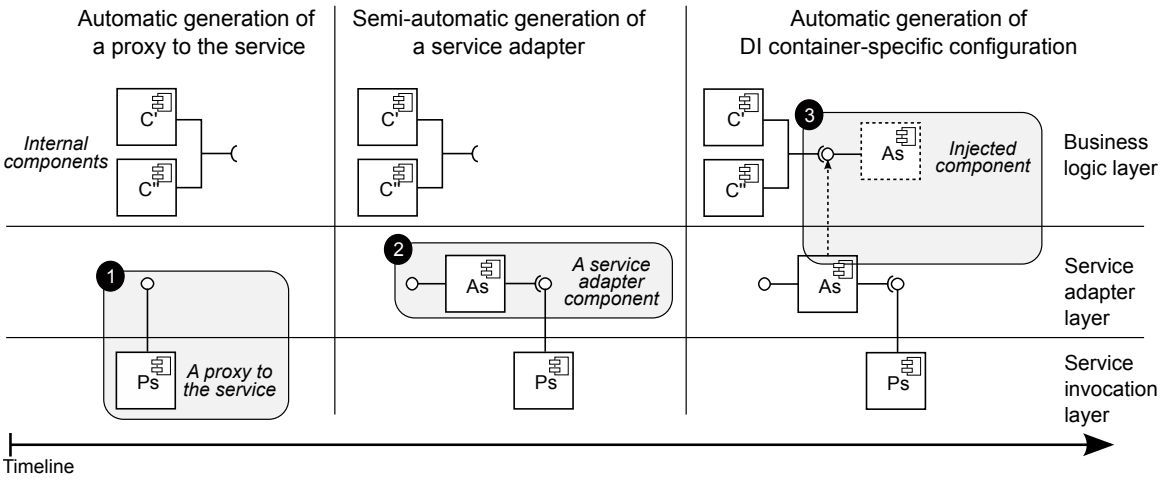


Figure 7: EasySOC steps for outsourcing services.

The current implementation of the Service Adapter module uses the Axis2 Web Service library for building service proxies, and Spring as the container supporting DI. Building a proxy with Axis2 involves giving as input the interface description of the target service (a WSDL document) to a command line tool. To setup the DI container, the names of dependant components and services must be written in an XML file. For adapting external service interfaces to the expected ones, we have designed an algorithm based on the work published in [21].

Our algorithm takes two Java interfaces as input, one is the Web Service interface and the other is the interface that the client application expects, and returns the Java code of a service adapter. To do this, it starts by detecting to which operations of the Web Service interface should be mapped the operations offered by the other. The algorithm assesses operation similarity by comparing operation names, comments, data-types and argument names. Data-type similarity is based on a pre-defined similarity table that assigns similarity values to pairs of simple data-types. The similarity between two complex data-types is calculated in a recursive way. Once a pair of operations has been determined, service adapter code is generated. The algorithm adapts simple data-types by taking advantage of type hierarchies and performing explicit conversions, i.e. castings. Complex data-types are resolved recursively as well. Clearly, not all available mismatches can be covered by the algorithm. Therefore, developers should revise the generated code.

In order to quantify the source code quality resulting from employing our plug-in, we conducted a comparison with the more traditional way of consuming Web Services, in which coding the application logic comes after discovering and knowing the description of the external services to be consumed. Basically, we used these two alternatives for developing a simple, personal agenda by outsourcing services from a given data-set comprising several services offering similar functionality but exposed by different providers.

After implementing the two variants, we randomly picked one service already included in the applications and we changed its provider. Then, we took metrics on the resulting source codes to have an assessment of the benefits of the EasySOC guidelines for software maintenance with respect to the traditional approach. To this end, we employed the well-known Source Lines Of Code (SLOC), Efferent Coupling (Ce), Coupling Between Objects (CBO) and Response For Class (RFC) software engineering metrics.

Table 4: Personal agenda: Source code metrics.

Variant		Id	SLOC	Ce	CBO	RFC
Initial Web Service providers	Traditional	T_1	242	7	4.50	30.00
	EasySOC	E_1	309	7	1.70	7.20
Alternative Web Service providers	Traditional	T_2	246	10	4.67	22.67
	EasySOC	E_2	327	10	2.00	7.45

Table 4 shows the resulting metrics values for the four implementations of the personal agenda: traditional, EasySOC, and two additional variants in which another provider for a service was chosen from the Web Service data-set. For convenience, we labeled each implementation with an identifier (*id* column), which will be used through the rest of the paragraphs of this section. To perform a fair comparison, a uniform formatting standard for all source codes was employed, Java import statements within compilation units were optimized, and the same tool to generate the underlying Web Service proxies was used.

From Table 4, it can be seen that the variants using the same set of service providers resulted in equivalent Ce values: 7 for T_1 and E_1, and 10 for T_2 and E_2. This means that the variants generated via our plug-in (E_x), did not incur in extra efferent couplings with respect to the traditional variants (T_x). Moreover, if we do not consider the corresponding service adapters, Ce for the EasySOC variants drops down to zero, because relying on EasySOC effectively push the code that depends on service descriptions out of the application logic. Interestingly, the lower the Ce value is, the less the dependency between the application code and the Web Service descriptions is, which simplifies service replacement.

Figure 8 shows the resulting SLOC. Changing the provider for a random service caused the modified versions of the application to incur in a little code overhead with respect to the original versions. The non-adapter classes implemented by E_1 were not altered by E_2 at all, whereas in the case of the traditional approach, the incorporation of the new service provider caused the modification of 17 lines from T_1 (more than 7% of its code).

The variants coded under EasySOC had an SLOC greater than that of the traditional variants. However, this difference was caused by the code implementing service adapters. In fact, the non-adapter code was smaller, cleaner and more compact because, unlike its traditional counterpart, it did not include statements for importing/instantiating proxy classes and handling Web Service--specific exceptions. Additionally, there are positive aspects concerning service adapters and SLOC. A large percentage of the service adapter code was generated automatically, which means programming effort was not required. Besides, changing the provider for the target service triggered the automatic generation of a new adapter skeleton, kept the application logic unmodified, and more importantly, allowed the programmer to focus on supporting the alternative service description only in the newly generated adapter class. Conversely, replacing the same service in T_1 involved the modification of the classes from which the service was accessed (i.e. statements calling methods or data-types defined in the service interface), thus forcing the programmer to modify more code. In addition, this practice might have introduced more bugs into the already built and tested application.

CBO and RFC metrics were also computed (Figure 9). Particularly, high CBO is undesirable, because it negatively affects modularity and prevents reuse. The larger the coupling between classes, the higher the sensitivity of a single change in other parts of the application, and therefore maintenance is more difficult. Hence, inter-class coupling, and specially couplings to classes

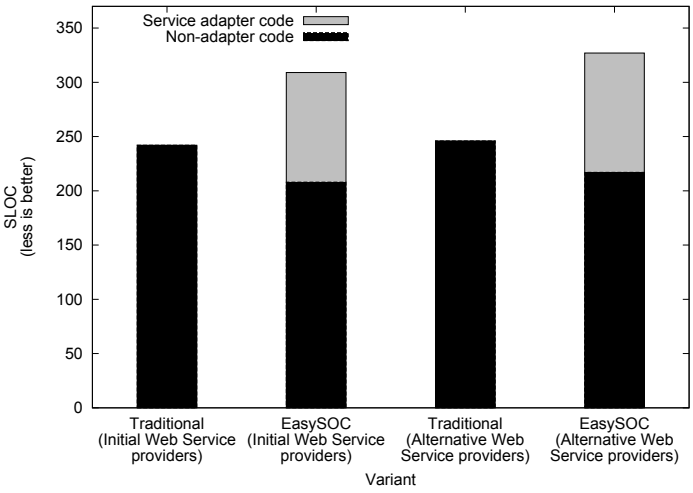
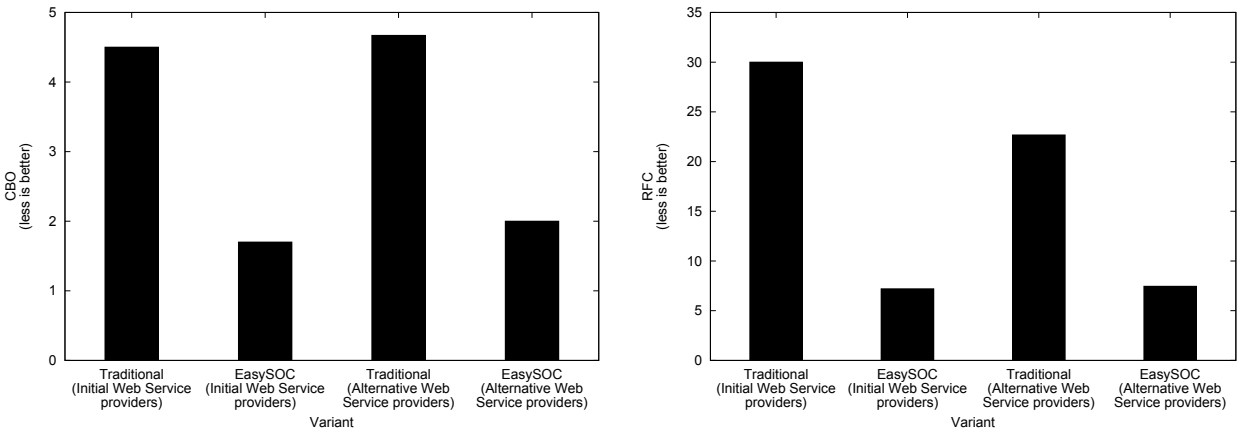


Figure 8: Source Lines of Code of the different applications.

representing (change-prone) service descriptions, should be kept to a minimum. Similarly, low RFC implies better testability and debuggability. In concordance with Ce, which resulted in greater values for the modified variants of the application, CBO for both the traditional approach and EasySOC exhibited increased values when changing the provider for a service. RFC, on the other hand, presented a less uniform behavior.



(a) CBO

(b) RFC

Figure 9: Coupling Between Objects and Response For Class of the different applications.

As quantified by Ce, EasySOC did not reduce the amount of efferent couplings from the package implementing the application logic. Naturally, the reason of this is that the service descriptions to which E_x adhere are exactly the same as T_x. However, the EasySOC applications reduced the CBO with respect to the traditional implementations, because the access to the various services utilized by the application, and therefore their associated data-types, is performed within several cohesive compilation units (i.e. adapters) rather than within few, more generic classes. This in turn improves reusability and testability since application logic classes do not directly depend on services.

As depicted in Figure 9 (b), this separation also helped in achieving better average RFC. Moreover, although the plain sum of the RFC values of the E_x were greater compared to T_x, the total RFC of the classes implementing application logic (i.e. without taking into account adapter classes) were both smaller. This suggests that the pure application logic of E_1 and E_2 is easier to understand. In large projects, we reasonably argue that much of the source code of EasySOC applications will be application logic instead of service adapters. Therefore, preserving the understandability of this kind of code is crucial.

4 Related work

This work is somehow related to a number of preliminary methodologies that have emerged to address the demand for process guidance for SOC. These methodologies build upon existing techniques, such as EA and BPM, but also agile processes, such as XP and RUP [22]. The most exhaustive and complete survey on approaches to SOC-based development is the work by Kohlborn and his colleagues [23], which in turn builds upon similar previous works [22, 24]. The authors have reviewed and compared 30 service engineering methods according to several dimensions, including *SOA (Service-Oriented Architecture) concept*, i.e. whether business-level and IT-level services are supported, *life-cycle coverage*, or the amount of development phases that are supported, and *accessibility and validity*, i.e. whether such efforts are well-documented and empirically validated, respectively. A business-level service is a set of actions that are performed by and reflect the actual operations of an organization. While IT-level services represent parts of a software system which can be consumed separately and support the execution of business services.

Although the reviewed methods are mostly aimed at providing guidelines at the development process level, and our work does not represent a development methodology per se, the guidelines proposed in this paper are somehow related to these methods in various respects. With regard to the SOA concept dimension, Kohlborn et al. conclude that up to 27 approaches provide support for IT-level or software services, only 8 methods are properly documented or publicly available, and the common approach to validation is through examples and case studies. Instead, we provide good practices for implementing loosely-coupled applications and software services. Therefore, unlike efforts such as [25, 26], we do not address materialization of business services.

With respect to the life-cycle coverage dimension, as our work prescribes well-defined steps for designing services and applications, it complements the existing methods. Specifically, we offer some proven guidelines for deriving understandable and search-effective Web Service descriptions during the service design phase. In addition, we provide guidelines for not only discovering a suitable service, but also decoupling the application for the particular service that it is consuming, which facilitates service replacement. Thus, the guidelines cover the development and maintenance phases of the applications.

Moreover, with respect to the accessibility dimension, we aim at making our best practices fully available in order to allow the SOC community and the software industry to exploit the proposed catalog of best practices, and to provide a tool set for materializing it. The tool set is publicly available for download at the project's Web site (`http://sites.google.com/site/easysoc`).

Regarding the validity dimension, it is worth remarking that the proposed guidelines have been followed to produce SOC-based software by playing both consumer and provider roles, thereby the collected empirical evidence supports that the proposed guidelines are indeed best practices. This aspect also makes our work differ from the WS-I Basic Profile [27], an industrial effort from the Web Services Interoperability Organization that comprises guidelines for structuring SOAP messages and WSDL documents according to well-defined rules. Besides, the WS-I Basic Profile puts emphasis on interoperability of Web Services and applications, while EasySOC focuses on improving discoverability and maintainability. Therefore, our guidelines and WS-I Basic Profile might be seen as complementary guidelines for Web Service design and implementation. Additionally, another point of difference between our work and efforts like [26] is that, at least to the best of our knowledge, EasySOC is the first attempt towards a tool-aided *step-by-step* guide for materializing both Web Services and client applications.

5 Conclusions

The software industry is embracing the SOC paradigm as the premier approach for achieving integration as well as interoperability in heterogeneous, distributed computing environments. However, SOC presents many intrinsic challenges that both Web Service providers and consumers must unavoidable face.

Historically, catalogs of best practices have been widely recognized as a very valuable and helpful mean for software practitioners to deal with common problems in many different contexts. In this sense, this paper presented a set of concrete, proven guidelines for avoiding recurrent problems when developing Web Services and client applications. The proposed catalog is a set of 3 guidelines comprising 16 steps. One guideline is intended for allowing potential consumers to effectively discover, understand and access them. This guideline has two sub-guidelines. Firstly, one sub-guideline covers 6 steps that service providers should take into consideration when exposing their services using the contract-first method. Secondly, another sub-guideline presents 4 steps to be followed when using code-first. The other two guidelines cover aspects that service consumers should consider when discovering and consuming services. Regarding discovery, the proposed guideline consists of 3 steps that should be followed to easily build effective queries, which alleviates consumers' task by narrowing down the number of candidate services. With respect to service consumption, following the 3 steps of the associated guideline results in more maintainable code within client applications, but also in those services that invoke other services to accomplish their tasks.

The practical implications of each guideline have been corroborated experimentally, which suggests that the guidelines can be conceived as being best practices and can be readily employed in the software industry. In particular, we have assessed the impact of removing WSDL discoverability anti-patterns from Web Service descriptions, by employing three registries simultaneously supporting service discovery and human consumers, who had the final word on which service is more appropriate. Results showed that improved descriptions are easier to understand than their "raw" counterpart [10]. Similarly, the positive effect on service discovery

of the guideline for generating and expanding queries has been also measured [6]. Also, the implications on clients' maintenance of the corresponding guidelines have been formally and experimentally shown in [7] and [8] respectively.

Clearly, building truly loose coupled client applications using the corresponding guidelines imposes a radical shift in the way such applications are developed. This means that a company willing to employ EasySOC to start producing service-oriented applications would have to invest much time in training its development team, which results in a costly start-up curve. The impact of EasySOC on the software development process itself from an engineering point of view has been empirically assessed in [9]. Concretely, we performed further experiments to test the following hypothesis that understanding pervasive design patterns (i.e. Adapter and DI) and the "first build your application and then servify it" philosophy are the only required intellectual activities to start developing service-oriented applications with EasySOC, which should sharpen the associated learning curve. The hypothesis has been confirmed with 45 postgraduate and undergraduate students of the Systems Engineering program at the UNICEN during 2009. Results showed that they perceived that the proposed approach is convenient and easily to adopt.

In the near future, we will conduct experiments with other students and real development teams to further validate our claims.

Acknowledgments

We thank the SCCC'10 chair Dr. Sergio F. Ochoa and the anonymous referees for their helpful comments to improve the paper. We also acknowledge the financial support provided by ANPCyT through grant PAE-PICT 2007-02311.

References

[1] J. Erickson and K. Siau, "Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality," *Journal of Database Management*, vol. 19, no. 3, pp. 42–54, 2008.

[2] M. Campbell-Kelly, "The rise, fall, and resurrection of software as a service," *Communications of the ACM*, vol. 52, no. 5, pp. 28–30, 2009.

[3] W3C Consortium, "SOAP version 1.2 part 1: Messaging framework," W3C Recommendation, http://www.w3.org/TR/soap12-part1, Jun. 2007.

[4] T. Erl, *SOA Principles of Service Design*. Prentice Hall, 2007.

[5] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising WSDL documents: Why and how," *IEEE Internet Computing*, vol. 14, no. 5, pp. 30–38, 2010.

[6] M. Crasso, A. Zunino, and M. Campo, "Combining query-by-example and query expansion for simplifying Web Service discovery," *Information Systems Frontiers*, 2009, to appear.

[7] C. Mateos, M. Crasso, A. Zunino, and M. Campo, "Separation of concerns in service-oriented applications based on pervasive design patterns," in *Web Technology Track (WT) - 25th ACM Symposium on Applied Computing (SAC '10)*. ACM Press, 2010, pp. 2509–2513.

[8] M. Crasso, C. Mateos, A. Zunino, and M. Campo, "EasySOC: Making Web Service outsourcing easier," *Information Sciences (SI: Applications of Computational Intelligence and Machine Learning to Software Engineering)*, 2010.

[9] C. Mateos, M. Crasso, A. Zunino, and M. Campo, "An evaluation on developer's acceptance of EasySOC: A development model for Service-Oriented Computing," in *11th Argentine Symposium on Software Engineering (ASSE2010) - 39th JAIIO*. SADIO, 2010.

[10] 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.

[11] R. A. Van Engelen and K. A. Gallivan, "The gsoap toolkit for web services and peer-to-peer computing networks," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 128–135.

[12] Apache Software Foundation, "Apache CXF: An Open Source Service Framework," http://cxf.apache.org, 2009.

[13] I. Androutsopoulos and P. Malakasiotis, "A survey of paraphrasing and textual entailment methods," *Journal of Artificial Intelligence Research*, vol. 38, pp. 135–187, May 2010.

[14] E. Al-Masri and Q. H. Mahmoud, "WSB: A broker-centric framework for quality-driven web service discovery," *Software: Practice and Experience*, vol. 40, pp. 917–941, Sep. 2010.

[15] E. Al-Masri and Q. H. Mahmoud, "Qos-based discovery and ranking of Web Services," in *16th International Conference on Computer Communications and Networks (ICCCN'07)*, 2007, pp. 529–534.

[16] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] J. Pasley, "Avoid XML schema wildcards for Web Service interfaces," *IEEE Internet Computing*, vol. 10, no. 3, pp. 72–79, May-June 2006.

[18] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10), Cape Town, South Africa*. New York, NY, USA: ACM Press, 2010, pp. 475–484.

[19] S. Stigler, "Fisher and the 5% level," *Chance*, vol. 21, pp. 12–12, 2008.

[20] E. E. Allen and R. Cartwright, "Safe instantiation in generic java," *Science of Computer Programming*, vol. 59, no. 1-2, pp. 26 – 37, 2006, special Issue on Principles and Practices of Programming in Java (PPPJ 2004).

[21] E. Stroulia and Y. Wang, "Structural and semantic matching for assessing Web Service similarity," *International Journal of Cooperative Information Systems*, vol. 14, no. 4, pp. 407–438, 2005.

[22] E. Ramollari, D. Dranidis, and A. Simons, "A survey of service oriented development methodologies," in *2nd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2007)*, 2007.

[23] T. Kohlborn, A. Korthaus, T. Chan, and M. Rosemann, "Identification and analysis of business and software services - a consolidated approach," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 50–64, 2009.

[24] F. Kohlmann and R. Alt, "Business-driven service modelling - a methodological approach from the finance industry," in *1st International Working Conference on Business Process and Services Computing (BPSC'07)*, ser. Lecture Notes in Informatics, W. Abramowicz and L. Maciaszek, Eds., vol. 116. GI, 2007, pp. 180–193.

[25] D. Flaxer and A. Nigam, "Realizing business components, business operations and business services," in *IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-EAST'04)*. IEEE Computer Society, 2004, pp. 328–332.

[26] M. Papazoglou and W.-J. van den Heuvel, "Service-oriented design and development methodology," *International Journal of Web Engineering and Technology*, vol. 2, no. 4, pp. 412–442, 2006.

[27] R. Chumbley, J. Durand, G. Pilz, and T. Rutt, "Basic profile version 2.0," http://ws-i.org/profiles/BasicProfile-2.0-WGD.html, Mar. 2010.