Best Practices for Describing, Consuming, and Discovering Web Services: A Comprehensive Toolset

Journal:	*Software: Practice and Experience*
Manuscript ID:	SPE-11-0098.R2
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	14-Mar-2012
Complete List of Authors:	Rodriguez, Juan; ISISTAN - UNCPBA, Computación y Sistemas Crasso, Marco; ISISTAN - UNCPBA, Computación y Sistemas Mateos Diaz, Cristian; ISISTAN - UNCPBA, Computación y Sistemas Zunino, Alejandro; ISISTAN - UNCPBA, Computación y Sistemas
Keywords:	SERVICE-ORIENTED COMPUTING, SERVICE-ORIENTED DEVELOPMENT GUIDELINES, WEB SERVICES, WSDL ANTI-PATTERNS, WEB SERVICE DISCOVERY, WEB SERVICE CONSUMPTION

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Best Practices for Describing, Consuming, and Discovering Web Services: A Comprehensive Toolset

Juan Manuel Rodriguez, Marco Crasso, Cristian Mateos*, Alejandro Zunino

ISISTAN Research Institute - UNICEN University
Tandil (B7001BBO), Buenos Aires, Argentina.
Tel./Fax: +54 (2293) 43-9682 ext. 35/43-9681
Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

SUMMARY

The Service-Oriented Computing (SOC) paradigm has recently gained a lot of attention in the software industry, since SOC represents a novel and a fresh way of architecting distributed applications. SOC is usually materialized via Web Services, which allows developers to structure applications exposing a clear, public interface to their capabilities. Although conceptually and technologically mature, SOC still lacks adequate development support from a methodological point of view. In this paper, we present the EasySOC project, a set of guidelines to simplify the development of service-oriented applications and services. EasySOC is a synthesized catalog of best SOC development practices that arises as a result of several years of research in fundamental Services Computing topics, i.e. WSDL-based technical specification, Web Service discovery, and Web Service outsourcing. In addition, we describe a materialization of the guidelines for the Java language, which has been implemented as a plug-in for the Eclipse IDE. We believe that both the practical nature of the guidelines and the availability of this software that enforces them may help software practitioners to rapidly exploit our ideas for building real SOC applications. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: 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, 2] 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 delivered and remotely consumed using standard protocols. Service-oriented software systems started as a more flexible and cost-effective alternative for developing distributed applications. Since SOC is platform-independent, it has enforced the development of software that interacts with third-party software. Furthermore, SOC systems might interconnect different organization systems. Finally, SOC usage eventually spread to gave birth to a wave of contemporary infrastructures and notions including Service-Oriented Grids [3] and Software-As-A-Service [4].

The common technological choice for materializing the SOC paradigm is Web Services, i.e. programs with well-defined interfaces that can be located, published and consumed by means of ubiquitous Web protocols [2] (e.g. SOAP [5]). The canonical model underpinning Web Services

*Correspondence to: Cristian Mateos, cmateos@conicet.gov.ar 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 an interface in Web Service Description Language (WSDL)[†]. WSDL is an XML-based language that allows providers to specify their services' functionality as a set of abstract operations with inputs and outputs, and to specify the associated binding information so that consumers can consume the offered operations. Inputs and outputs have associated data-types that are specified in XSD (XML Schema Definition) [6], an extensible data-type specification language. There are two approaches to build WSDL documents. WSDL-last or code-first refers to infer WSDL documents from services implementations. Contrarily, WSDL-first or contract-first means building WSDL documents from scratch, and in turn supplying them with implementations thereafter.

Basically, WSDL-last requires no knowledge about WSDL or XSD, since WSDL documents are not generated by humans and these are automatically extracted from services implementations. Yet, the developer has no control on the resulting WSDL documents so new different deployments of the same service might be incompatible [7]. In addition, the resulting WSDL documents might be difficult to be read and used by third-parties [8]. In contrast, WSDL-first requires developers skilled in WSDL and spending time to actually write the WSDL document introducing new costs. However, these WSDL documents tend to be more interoperable and present better quality [9]. Therefore, both approaches have pros and cons, making impossible to claim that choosing one over the other will be always a better decision [7].

To make their WSDL documents publicly available, providers originally employed a specification of service registries called UDDI, whose central purpose is the representation of meta-data about Web Services [10]. However, practitioners have not massively adopted UDDI because its inherent limitations. Basically, UDDI registries only allow to search service via keywords, which is not appropriate when the number of registered services is very high [11]. Therefore, several researchers have proposed new approaches to facilitate service discovery [12, 13]. These approaches can be divided into two types: semantic based and information retrieval based.

Semantic based registries promise the ability of allowing consumers to look for a service according to their functionality, thereby semantic based registries would always retrieve services that fulfill the consumers' needs [12]. However, to publish a service in a semantic based registry, the service must be specified using some semantic annotation language, such as OWL, which implies an extra development effort. In addition, such specification must be based on a particular ontology, making even harder to publish the same service in different registries that do not share the ontology. As a result of these limitations, most of the publicly available services' providers does not annotate their service using semantics [14]. In contrast, information retrieval based registries can publish services without other specification but their WSDL documents. This kind of registries employs the WSDL documents to gather terms that might describe the associated services, and uses them to match services with service consumers' queries. Although information retrieval based registries might incur in some mismatches, they are effective enough to be a viable alternative to service discovery [12].

Unfortunately, the popular "there is no such thing as a free lunch" adage also applies in this context. 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 introduced by current approaches for implementing the SOC paradigm. On one hand, unless appropriately specified by providers, service meta-data can be counterproductive and obscure the purpose of a service, thus hindering its adoption. For example, a WSDL description without much comments 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 manually discovering Web Services (i.e. inspecting a UDDI registry or an information retrieval based one), and then providing code to consume them. Moreover, the outcome of the second task is software

[†]WSDL, http://www.w3.org/TR/wsdl

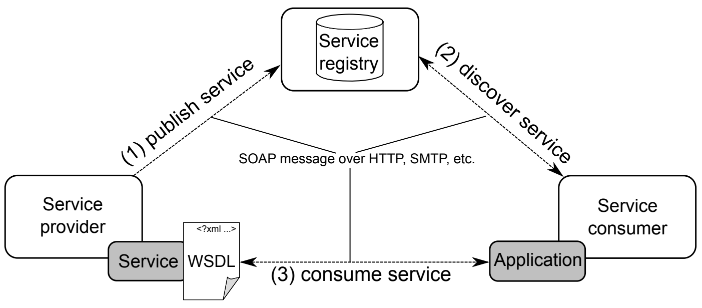


Figure 1. The Web Services model.

containing service-aware code, and therefore it is difficult to test and to modify during the software maintenance phase.

When a service consumer has selected the Web Service that fulfills their necessities, the Web Service is integrated into the system using code that it is aware of the service specific interface. This code is called glue code because it is for making compatible software components that are otherwise incompatible rather than implementing functionality. The problem of this glue code is that it is usually scattered across the system, making it difficult to exchange a service for another with the same functionality, but different logic [15]. This situation impacts negatively on the loose coupling feature promised by Web Services because it introduces contract-to-functional coupling [16], which means that service invocations depend on the service contract. As a result of these issues, Web Services have not been as heavily adopted as it was expected when the paradigm first appeared.

In this paper, we describe the EasySOC project, a set of provider and consumer guidelines for alleviating these problems. Roughly, these guidelines represent a compilation of best practices for simplifying the activities required to implement the SOC paradigm with the RPC model, which are illustrated by the arcs of Figure 1, while improving the quality of the artifacts implementing services and applications. Here, and throughout the paper, the term “application” refers to software that consumes one or more third-party services. EasySOC is based on extensive previous research carried out by the authors in the subareas of WSDL-based technical contract design and specification [17], Web Service discovery [18], applications development and maintenance [19], and developers' acceptance of these guidelines [20].

Complementary, this paper contribution is to provide a uniform, conceptualized and synthesized view of these findings to provide, on one hand, clear and precise hints of how to adequately exploit the SOC paradigm and its related technologies regardless their usage context, i.e. when implementing applications or services. At the same time, another contribution is to delineate potential concrete materializations 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, which helps developers to employ the guidelines.

This work is based upon [19, 21, 20], but this work focus is on how service developers can take advantage of the findings presented on these works. Briefly, this paper presents:

- a comprehensive set of guidelines for service providers and consumers,
- a toolset for enforcing the application of each guideline,
- empirical evidence of the effectiveness of the toolset that enforces the guidelines.

The rest of the paper is structured as follows. Section 2 surveys relevant related efforts. Section 3 focuses on discussing the aforementioned guidelines, emphasizing on clarifying their scope and the usage scenarios in which these guidelines are applicable. Later, Section 4 presents the EasySOC Eclipse plug-in, its modules and related experiments. Finally, Section 5 concludes the paper.

2. RELATED WORK

Certainly, SOC is a software engineering paradigm that introduces opportunities as well as challenges from a methodological perspective. Although traditional development processes and practices –e.g. XP, RUP– can be partially reused in or adapted for this new scenario, many researchers agree that novel techniques are needed to address the specifics of SOC requirements [22]. In this sense, work in this sub-area is quite active and mainly pursues one common goal, namely providing comprehensive process guidance, recommended practices and tool sets to do SOC.

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 technology-agnostic actions that are offered by an organization. A business-level service represents the actual operation 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.

Kohlborn et al. [23] identify two main phases to develop Web Services that represent a specific core business. Firstly, the authors indicate that business-level services must be identified through documentation, and interviews with the organization's senior members. This helps to identify, prioritize, and delimit which are the services that best represent the purpose of the organization. Secondly, these business-level services are mapped to IT-level services. During this stage, Web Services are designed, associating one or more Web Service to business-level services. As a result of these steps, the SOA frontier of an organization can effectively represent the organization actions and objectives.

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 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 from 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 both services and applications, together with a rigorous experimental evaluation of their benefits. Therefore, the collected empirical evidence supports that the proposed guidelines are indeed best practices. Note that our guidelines do not cover all the aspects of Web Services, and thereby other guidelines such as WS-I Basic Profile [27] must be taken into account by service providers and consumers. WS-I Basic Profile is an industrial effort from the Web Services Interoperability Organization that comprises guidelines for structuring SOAP messages and WSDL documents according to well-defined rules. This is, WS-I Basic Profile puts emphasis on interoperability of Web Services and applications at the service specification and communication levels, while EasySOC guidelines aim at providing interoperability at the business level by addressing service replacement in a technology-neutral way, yet the tool is implemented for JAVA platform. In this context, interoperability at the business level means the capability of being independent of specific service providers contract descriptions. Finally, EasySOC also focuses on improving discoverability and maintainability. Therefore, our guidelines and WS-I Basic Profile might be seen as equally important, complementary guidelines for Web Service design and implementation.

3. 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 service and application developers.

Schematically, according to the model of Figure 1, two distinctive, albeit non mutually exclusive 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. This application can also be exposed as a service in which case the service consumer becomes at the same time a service provider.

Sometimes the same developer can play both roles, as occurs when developing services that need other services to accomplish the functionality that they expose. In this way, depending on the role(s) played by a developer, we have identified three different scenarios. Table I lists these scenarios by relating them to the guidelines that developers are encouraged to pay attention to, and in which development process are this guidelines applicable. The guidelines contemplate the development scenarios of exposing a service, consuming a service, and exposing a service that consumes other services.

Table I. 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	When designing a service interface	Never	Never
Developer only consumes services	Never	When looking for a service	When consuming a service
Developer exposes as a service a functionality that consumes other services	When designing a service interface	When looking for a service	When consuming a service

To understand the implications of employing the proposed guidelines, a case study will be discussed through the rest of the section. The case study is a reference application provided by Sun to illustrate how developers can apply various Java Enterprise Edition technologies for implementing Web Services. The reference application consists of a Web-based application called Adventure Builder[‡]. The application provides customers with a catalog of adventure

[‡]Java Adventure Builder Reference application 1.0.1:http://java.sun.com/developer/releases/adventure/

packages, accommodations, and transportation options. Customers select these elements to build a vacation plan, such as mountain biking on Tandil. Building a vacation plan includes selecting accommodations, transport media and adventure activities. After assembling the vacation package, the customer builds a purchase order.

At the heart of the Adventure Builder application is the order processing center (OPC), a central component that coordinates external Web Services for fulfilling customer's orders. The OPC functionality for handling a single customer order is offered as a Web Service as well. The external Web Services consumed by the OPC offer operations for supplying airline tickets, hotels and adventure activities, and verifying/approving credit cards. Figure 2 depicts the Adventure Builder application component diagram using the UML 2.0 notation for modeling components.

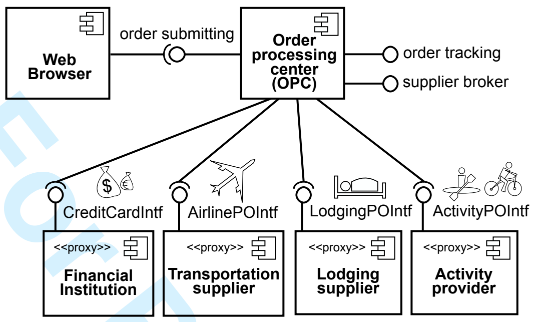

Web Browser
order submitting
Order processing center (OPC)
order tracking
supplier broker
CreditCardIntf
AirlinePOIntf
LodgingPOIntf
ActivityPOIntf
<<proxy>> Financial Institution
<<proxy>> Transportation supplier
<<proxy>> Lodging supplier
<<proxy>> Activity provider

Figure 2. Adventure Builder: Components.

This case study has been intentionally chosen since the development of the OPC and its Web Service interface requires to play both consumer and provider roles, respectively. Then, the Adventure Builder development covers all the scenarios of Table I.

3.1. Guidelines for improving service descriptions

A service life-cycle consists of several phases, from which the service design phase comprises the service interface specification using WSDL. Several important concerns, such as granularity, cohesion, discoverability, reusability, should influence design decisions to result in an efficient service interface design [26]. Many of the problems related to the efficiency of standard-compliant approaches to service discovery stem from the fact that the WSDL is incorrectly or partially exploited by providers. Despite the intuitive importance of properly describing services, some practices that negatively impact on services' discoverability, such as poorly commenting offered operations or using unintelligible naming conventions, are frequently found in publicly available WSDL documents [21].

In particular, we have detected eight different issues in real-life WSDL documents that might affect its discoverability and understandability [21]:

1. XSD definition embedded in WSDL documents.
2. Different port- types with the same set of operations.
3. Different data-types for representing the same object of the problem domain coexist in a WSDL document.
4. Using flexible data-type to represent any object of the problem domain.
5. Using output messages to notify service errors.
6. Using large, ambiguous or meaningless names for denoting the main elements of a WSDL document.
7. Port-types with weak semantic cohesion.
8. Having no comments, or comments being inappropriate and not explanatory.

Although no silver bullet can guarantee that potential Web Service consumers will effectively discover, understand and access it, we have empirically shown that WSDL documents can be improved to simultaneously address these issues by following a six step guide [17]:

1. Separating data-type definitions –i.e. XSD [6] code– from the definitions of the offered operations (related to Issue 1).
2. Removing repeated WSDL and XSD code (related to Issues 2,3 and 4).
3. Putting error information within Fault messages and only conveying operation result within Output ones (related to Issue 5).
4. Replacing WSDL element names with short self-explanatory names if they are inappropriate, or longer than 15 characters as proposed in [28] (related to Issue 6).
5. Moving non-cohesive operations from their port-types to a new port-type (related to Issue 7).
6. Properly commenting the operations (related to Issue 8).

The first step means to move complex data-type definitions into a separate XSD document, and to add the corresponding import sentence into the WSDL document. However, when data-types are not going to be reused they can be part of the WSDL document to make this latter "self-contained".

The second step deals with redundant code in both the WSDL document and the schema. Repeated WSDL code usually stem from port-types tied to a specific invocation protocol, whereas redundant XSD is commonly a result from data definitions bound to a particular operation. Yet, according to the WSDL standard, port-types are abstract elements, and the link between a port-type and a protocol defines a *binding*. Therefore, repeated WSDL code can be removed by defining a *protocol-independent port-type*, plus as many bindings as invocation protocols are supported by the service.

Similarly, to eliminate redundant XSD code, repeated data-types should be abstracted into a single one. This change must be consequently made visible in messages by updating their data-types in order to reference the newly derived types. Too generic type definitions –definitions that are based on xsd:any– should be avoided because they only obscure the semantic of the data. This problem is analogous to using too many generics types in Java, such as Object for defining the return class or parameters' classes of a method, to assure extensibility [29]. Besides, as these definitions allow all the other defined data, they are inherently redundant.

Although xsd:any type has been identified as a versioning and extending mechanism [16], this technique does not come without cost [30]. In addition to the lack of semantic information, using generic types increases the complexity of consuming the service. The consumer must know what the service expects or returns in all the messages that use that type. As a result of being generic, both the service consumer and the service provider must implement validation logic for messages conveying xsd:any data-types. Finally, this versioning strategy has no warranty of backward compatibility. For example, there is a service that retrieves a weather forecast that uses as input a location whose type is xsd:any. In its first version the service uses as input a city name, but the new version uses as input geographical coordinates. Although the service interface does not change, both service providers and consumers must change the interaction logic of their applications.

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

The fourth step aims to improve the expressivity 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 or data-type names should be a noun or a noun phrase because they represent the objects on which the operation executes. If those names represent actions, it is possible that the information conveyed in those messages modifies the behavior of the operation [21]. Additionally, the names should be written according to common notations, and their length should be between 3 and 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 on their functional 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. In this context, functional cohesion means that operation belonging to the different functional domain should be in different port-types. Having a high functional cohesion has been recognized as a very important quality factor since structured design appeared [31, 32]. Cohesion metrics are usually taken from the source code [33], thereby how the cohesion is measure depends on how the service is implemented.

Finally, the step 6 reflects the fact that 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.

Notice that except for step 6, the other steps might require to modify a service implementation. Moreover, as a result of applying these guidelines, there will be two versions of a revised service description. In some cases, this new service version may be backward compatible with the previous version, which means that service consumers would not need to change their software because the new service and the old one are consumed in the same way [16]. For example, separating the XSD definitions from the WSDL document would not affect how the service is called. However, other changes, such as renaming operations, might affect the way services are consumed. In this case, the commonest versioning strategy is to keep the old and the new version of the service running in parallel for a while, until service consumers can modify their systems. For instance, different versions of Google Adword WSDL[§] documents are placed in different URLs. It seems that Google's developers use a part of the URL as version indicator, namely /v201101/ and /v201008/, hence service consumers can determine which version they are using. These are not the only two possibilities. However, since WSDL documents were not designed to support versioning [34], service providers must define it own versioning policy [16].

In order to exemplify how to apply this guideline, we have selected a WSDL document from one of the services in the Adventure Builder Application. In particular, the selected WSDL document presents the description of the credit card validation service. The service offers one operation that validates a credit card. The left side of Figure 3 depicts the selected WSDL document.

Basically, the first step recommends separating the data-type definition from the WSDL document. In this case, the data-type definition is the WSDL document largest part, being more than the half of the document. To make the WSDL document conciser, data-type definition should be taken out from the WSDL document and placed in a separate XSD file.

Regarding the second step, no repeated WSDL code appears because the WSDL document defines one port-type with one operation that uses two different messages. In contrast, the data-type definition presents redundant code. For instance, when we analyzed the *validateCreditCard* data-type, we found that it defines a sequence –that specifies that the child elements must appear in a specific order– of only one element, which is of the *String* type. Instead, using the primitive *String* data-type, which is defined as part of the standard XSD, can simplify the data-type definition. An analogous case is the *validateCreditCardResponse* data-type, but in this case the primitive type concerns the *boolean* data-type. Another option for credit card data-type can be using a sequence with different elements, such as card holder, credit card number, and security code. This specification would allow being more specific in which information is required by the service. This option would also need to modify the service implementation.

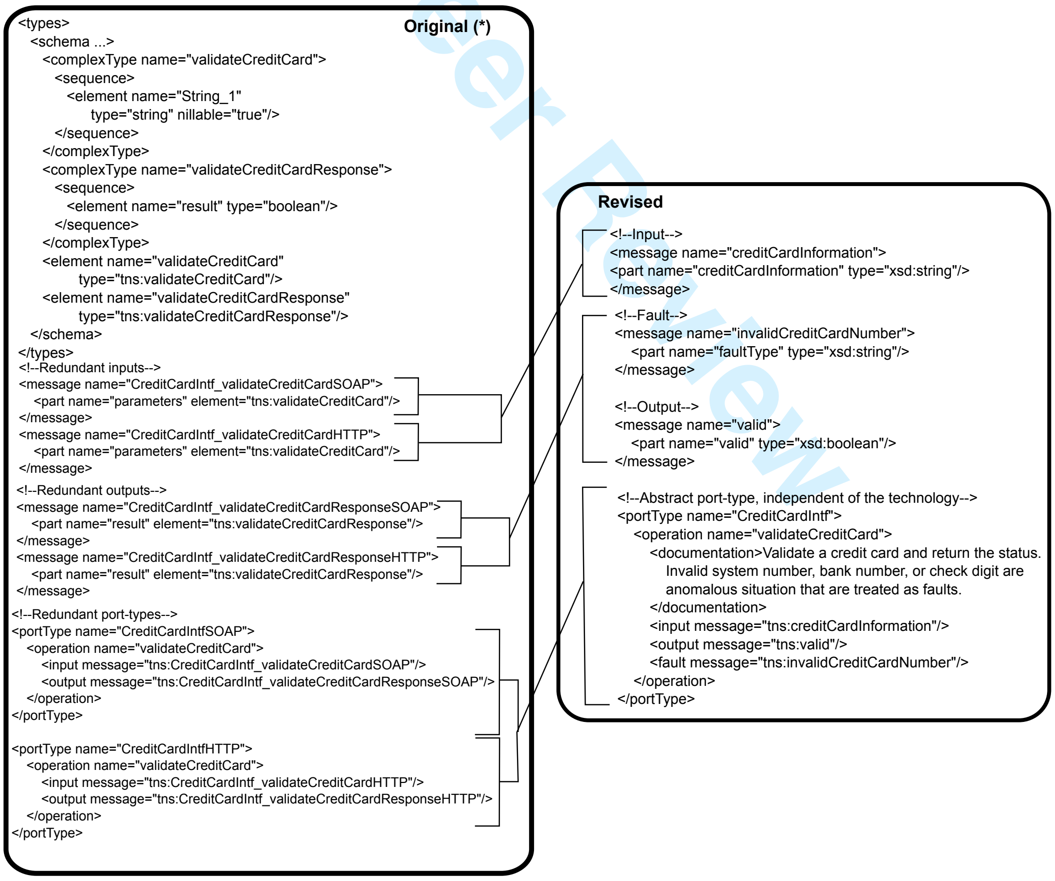
Apparently, at least from the WSDL document, no evidence indicates that the offered operation trigger any error. However, we assume that certain situations might trigger an error, thereby we have added a fault message to the operation. For the sake of the example, we have considered that malformed credit card numbers are the only possible fault cause. Yet, in a real life service there could be more faults caused by other problems, such as when a bank system is off-line.

[§]Google Adword v201101: https://adwords.google.com/api/adwords/cm/v201101/AdExtensionOverrideService?wsdl
Google Adword v201008: https://adwords.google.com/api/adwords/cm/v201008/AdExtensionOverrideService?wsdl

Clearly, message part names are related to the function of the message itself but not to the semantics of the carried information. This is because message parts are named after the operation and not after the information they convey. Besides, although the rest of the element names in the document are domain-specific, some of them are too long. That is the case of *CreditCardIntf_validateCreditCard*, which is the name of the input message. This makes names inappropriate (step 4) for the service consumer because such names have not semantic relationship with what they represent. Besides, too long names usually do not fit on screens and require scrolling to be read. In our example, a better name for the input message may be *creditCardInformation* or simply *creditCard*.

Since the port-type exposes one operation, all the operations in the port-type are semantically related (step 5). However, a commonly found example of this bad practice is the inclusion of operations, such as "isAlive", "getVersion" and "ping", within port-types that provide operations of a particular problem domain, e.g. to give information on commodities. This affects the Web Service because it not only makes it difficult for service consumers to understand why these operations are mixed up, but also introduces terms that do not describe the service when populated into a service registry. Therefore, such operations should be in a separate port-type.

Regarding the comments (step 6), the WSDL document has not comments, which results in hiding the true purpose of the service. For instance, service consumers might infer that the service functionality is validating credit cards; however, they are not able to know whether it validates cards by just checking if the number is suitable or additionally checking that the credit card is actually an authorized one. Even more, when the service returns false, service consumers are not able to know whether it means that the card number is not valid or could not be validated instead.



```
<types>
  <schema ...>
    <complexType name="validateCreditCard">
      <sequence>
        <element name="String_1"
            type="string" nillable="true"/>
      </sequence>
    </complexType>
    <complexType name="validateCreditCardResponse">
      <sequence>
        <element name="result" type="boolean"/>
      </sequence>
    </complexType>
    <element name="validateCreditCard"
          type="tns:validateCreditCard"/>
    <element name="validateCreditCardResponse"
          type="tns:validateCreditCardResponse"/>
  </schema>
</types>
<!--Redundant inputs-->
<message name="CreditCardIntf_validateCreditCardSOAP">
   <part name="parameters" element="tns:validateCreditCard"/>
</message>
<message name="CreditCardIntf_validateCreditCardHTTP">
   <part name="parameters" element="tns:validateCreditCard"/>
</message>
<!--Redundant outputs-->
<message name="CreditCardIntf_validateCreditCardResponseSOAP">
   <part name="result" element="tns:validateCreditCardResponse"/>
</message>
<message name="CreditCardIntf_validateCreditCardResponseHTTP">
   <part name="result" element="tns:validateCreditCardResponse"/>
</message>
<!--Redundant port-types-->
<portType name="CreditCardIntfSOAP">
   <operation name="validateCreditCard">
      <input message="tns:CreditCardIntf_validateCreditCardSOAP"/>
      <output message="tns:CreditCardIntf_validateCreditCardResponseSOAP"/>
   </operation>
</portType>

<portType name="CreditCardIntfHTTP">
   <operation name="validateCreditCard">
      <input message="tns:CreditCardIntf_validateCreditCardHTTP"/>
      <output message="tns:CreditCardIntf_validateCreditCardResponseHTTP"/>
   </operation>
</portType>
```

```
<!--Input-->
<message name="creditCardInformation">
<part name="creditCardInformation" type="xsd:string"/>
</message>

<!--Fault-->
<message name="invalidCreditCardNumber">
   <part name="faultType" type="xsd:string"/>
</message>

<!--Output-->
<message name="valid">
   <part name="valid" type="xsd:boolean"/>
</message>

<!--Abstract port-type, independent of the technology-->
<portType name="CreditCardIntf">
   <operation name="validateCreditCard">
      <documentation>Validate a credit card and return the status.
         Invalid system number, bank number, or check digit are
         anomalous situation that are treated as faults.
      </documentation>
      <input message="tns:creditCardInformation"/>
      <output message="tns:valid"/>
      <fault message="tns:invalidCreditCardNumber"/>
   </operation>
</portType>
```

(*) The original WSDL document has been adapted for the sake of exemplification

Figure 3. Credit card validation: original and revised WSDL documents.

The right side of Figure 3 depicts the improved WSDL document that resulted after splitting the WSDL document into a new XSD file and the document itself, and then removing the redundant data-types and reference to the primitive data types, which resulted in an empty XSD file. As a consequence of these actions, the size of the WSDL document has reduced to half of the original size. Moreover, the first consequence would stand even if the data-types were not redundant because of the separation of the WSDL document and its XSD definitions.

Regarding names, in the revised WSDL document we have included shorter names that additionally keep the modeled semantics, and as such are easier to read by service consumers. On the other hand, we added comments informing that the service is actually a stub for testing purposes. Therefore, service consumers are now aware that the result returned by the service is always true independently of the input it has receives.

3.2. Guidelines for making effective queries

During the life-cycle of an application, specifically in the process of service discovery, queries play an important role since service consumers may greatly benefit from generating clear and explanatory descriptions of their needs. In general, the more descriptive queries are, the more accurate discovery results they achieve. Moreover, as the underpinnings of UDDI-based registries rely upon the descriptiveness of the keywords conveyed in interfaces of publicly available services and in queries [10, 11, 21], consumers should pay special attention to which keywords they use. Then, the generation of effective queries presents two issues:

1. There is a correlation between consumers' search effort and the benefit they may obtain from a discovery system,
2. The employed keywords are important, thus they should carefully chosen.

The source code artifacts of applications may carry relevant keywords about the functional descriptions of the potential services that can be discovered and, in turn, consumed from within these applications. We have empirically proved that those keywords included in the source code can be *automatically* gathered and used as queries when some steps upon building applications are followed [18]. In this line, best practices for constructing an application such that it contains useful keywords about the external services it needs comprise:

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

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 [18]. This approach allows a consumer 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 applications and Web Service descriptions in WSDL, the "shape" of 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 applications, the defined interfaces stand for *examples*.

The second step bases on an approach for automatically augmenting the quantity of relevant keywords within queries, called *Query Expansion*. This approach relies on the expansion of extracted examples by gathering keywords from the source code representing internal components that directly interact with the interfaces representing external services [18]. The reasoning that supports this mechanism is that expanding queries based upon components with strongly-related and

Listing 1: Interface Validator

```java
public interface Validator{
 boolean validate(Object cc);
}
```

Listing 2: Interface ICreditCardValidator

```java
public interface ICreditCardValidator{
 /**
  * Validates a credit card
  */
 boolean validateCard(CreditCard creditCard);
}
```

highly-cohesive operations should not only preserve, but also improve the meaning of the original request or 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 parts of the source code of an application that may contain relevant keywords for generating queries and discovering Web Services afterward. Clearly, these two steps can be performed in many different ways, however not every alternative would be the best for making effective queries. In order to illustrate this, we will implement the OPC component of the described reference application. According to step 1, the developer of the OPC should define interfaces for the external functionalities needed. For the case of the credit card validation, the developer should define an interface exposing one operation that returns whether a credit card is valid or not. Following step 2, as the OPC is the only internal component that directly interacts with the credit card validation functionality, the developer should revise the functional cohesion between the OPC and the credit card validation component, which in this case is adequate. Concretely, listings 1 and 2 present two different alternatives for defining such interface.

The first defined interface, which is named Validator, conveys only two relevant keywords to derive a Web Service query, i.e. "`Validator`" and "`validate`". This is because the Java reserved words `public`, `interface`, and `boolean` must be discarded, and the term "`Object`" representing the data-type of the parameter is too general and also have a low level of usefulness within the context of credit card validation.

At the same time, the interface named ICreditCardValidator conveys in principle the keywords: `Credit`, `Card`, `Validator`, `Validates`, `credit`, `card`, `validate`, `Card`, `Credit`, `Card`, `Credit` and `Card`. After properly cleaning this list, we end up with less nevertheless descriptive terms to query against service registries.

In the example, two interfaces ranging from a poorly described interface to a good described were presented. Although the example is rather simple, it allows us to show that the way an interface is defined directly impacts in the quantity and quality of keywords that can be gathered from it, which was empirically confirmed in [18]. Thus, the guidelines for performing service queries propose a third step for checking that a defined interface conveys in its source code as more explanatory keywords as possible. As such, this third step indirectly encourages developers to follow best practices for naming and commenting their source code.

3.3. Guidelines for shielding applications from service specifics

As the development of an application moves through its life-cycle, maintenance becomes more important. Early design decisions can severely impact on the maintenance phase. Maintaining applications can be a cumbersome task when they are tied to specific providers and WSDL documents. The common approach to call a Web Service is by interpreting its associated WSDL document with the help of invocation frameworks such as WSIF [35], CXF [36], or the Windows

Communication Foundation[¶]. These frameworks succeed in hiding some of the details for invoking services, but they 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, such as service location, operation signatures, and data-types. This approach leads to code that is subordinated to third-party service interfaces and must be modified and/or re-tested every time a provider introduces any changes, which hinders service replaceability. To sum up, there are some issues related to service incorporation:

1. Application designs are tied to specific service interfaces,
2. Business logic code is contaminated with boilerplate code.

We have shown that the maintenance of service-oriented code can be facilitated by following certain programming practices when outsourcing services [19]:

1. Defining the expected interface of every component that is planned to be outsourced (related to Issue 1).
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 (related to Issue 1).
3. Injecting adaptation code into each internal component that depends on the expected interface (related to Issue 2).

Step 1 provides a mean for shielding the internal components of an application from the mentioned details related to third-party services consumption. 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 3.2. 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 candidate 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 the interface defined previously. For instance, if a service operation returns a list of integers, but the previously defined interface operation returns an array of floats, the developer should code a *service adapter* that performs the type conversion. By properly accomplishing steps 1 and 2, internal components depend on neither specific service implementations nor interfaces.

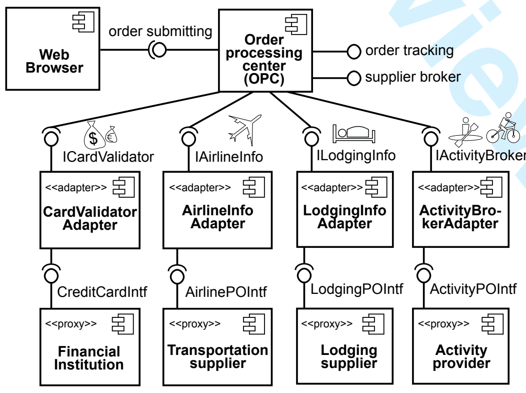

Web Browser
order submitting
Order processing center (OPC)
order tracking
supplier broker
ICardValidator
IAirlineInfo
ILodgingInfo
IActivityBroker
<<adapter>> CardValidator Adapter
<<adapter>> AirlineInfo Adapter
<<adapter>> LodgingInfo Adapter
<<adapter>> ActivityBrokerAdapter
CreditCardIntf
AirlinePOIntf
LodgingPOIntf
ActivityPOIntf
<<proxy>> Financial Institution
<<proxy>> Transportation supplier
<<proxy>> Lodging supplier
<<proxy>> Activity provider

Figure 4. Adventure Builder: Components plus Adapters.

This is graphically shown in Figure 4, where after accommodating the design of the Adventure Builder application according with steps 1 and 2, the OPC depends on the interfaces ICardValidator, IAirlineInfo, IActivityBroker, and ILodgingInfo. The bindings between such interfaces and the

[¶]Windows Communication Foundation, http://msdn.microsoft.com/en-us/netframework/aa663324

Listing 3: Method validateCreditCard

```
public boolean validateCreditCard(String ccXMLString)
               throws RemoteException;
```

Listing 4: OPC code for calling a Web Service

```
...
CreditCardService ccSvc = (CreditCardService)
 JNDI_lookup("java:comp/env/service/CreditCardService");
CreditCardIntf port = ccSvc.getPort(CreditCardIntf.class);
...
String creditCardXML = po.getCreditCard().toXML();
boolean ccStatus = port.verifyCreditCard(creditCardXML);
...
```

actual service interfaces are iteratively materialized at implementation time via service adapters. Accordingly, changes in the consumed services only require modifying the service adapters, leaving the OPC unaffected. Therefore, from the perspective of the application logic, services that provide equivalent functionality can be transparently interchangeable.

Finally, step 3 is for separating the functional code of an application from configuration concerns related to binding an internal component that depends on an interface, with the component in charge of adapting it into a selected service. A suitable form of doing this, in terms of source code quality, involves delegating the administrative task of assembling interfaces, internal components and services to a software layer or container.

To illustrate these steps, let us consider the implementation of the interaction between the OPC and the credit card validation service in the Adventure Builder application. As mentioned, this Web Service implements only one operation, whose signature and implementation are shown in listings 3 and 4 respectively, that returns whether a credit card is valid or not by taking as input a bank-specific XML representation of a credit card. Moreover, from the OPC component, the Web Service is accessed as shown in listing 6, where po is the object representing a customer's purchase order and provides access to an object containing the credit card information. This latter is implemented in such a way that it knows how to generate the XML representation of the card information expected by the validation service.

The source code presented in listing 4, however, presents several drawbacks, which directly affect flexibility and maintainability. First, it contains statements that explicitly depend on the binding-specific technology (JNDI) and information (URIs, exception classes) used. Second, it depends on the interface and hence the operations of this particular validation Web Service. Third, it is highly coupled not only to the data-types expected by the service –due to the previous drawback– but also to the data representation that is accepted by the service. In the event of changing the provider for the validation service, which is a common scenario in SOC development, significant portions of this source code must be rewritten.

To avoid these problems, as suggested earlier, we should first to define the ICardValidator interface, which is shown in Section 3.2. The ICardValidator interface stands for the *potential* but not the real interface of existing validation services. The ICardValidator interface includes an operation that receives a CreditCard business object, which not include neither information nor behavior –e.g. a toXML() method– specific to a particular Web Service. At step 2, such details are precisely hidden (see listing 5) from the components accessing the Web Service behind a service adapter.

Listing 5: Adapter Class

```
class CardValidatorAdapter implements ICardValidator{
 CreditCardIntf port = null;
 public CardValidatorAdapter(){
  // Instance-specific binding information
  CreditCardService ccSvc = (CreditCardService)
```

Listing 6: Calling the adapter

```
...
ICardValidator port = getValidator();
...
boolean ccStatus = port.verifyCard(po.getCreditCard());
...
```

```
    JNDI_lookup("java:comp/env/service/CreditCardService");
   // Instance-specific service interface
   port = ccSvc.getPort(CreditCardIntf.class);
  }
  public boolean validateCard(CreditCard creditCard){
   try{
    return port.verifyCreditCard(toXML(creditCard));
   }
   catch(RemoteException exe){
    System.err.println(exe);
    return false;
   }
  }
  // Instance-specific service data representation
  private String toXML(CreditCard creditCard){...}
}
```

In short, the adapter concentrates the details of particular instances of validation services under an umbrella given by a common, clear interface. Lastly, the source code of the OPC component must be modified to use the adapter as it is shown in listing 6.
Consequently, the logic of OPC is decoupled from the validation service. Moreover, getValidator is a tool-injected getter that returns an instance of the adapter. Analogously, there is another injected method to instantiate the adapter at run-time. This can be supported by using existing component assembling techniques for keeping coupling between business components and adapters low, such as Dependency Injection [37], which is in fact used in our current software materialization. Conceptually, this technique promotes separation of business logic from details associated with external services. In this sense, the same idea is applied to make adapters such as CardValidatorAdapter to be independent of the binding information of the adapted service. One might want, for example, to use another URI or even information that results from employing a different discovery technology.

A drawback of this approach is that the adapter must be reimplemented when selecting a new service, thereby it is not possible to change the service for a new one dynamically. Although it is a well-known SOA promise, this is not true even with different version of the same service [34]. Simply changing an operation name might render a service unusable by those service consumers that were already using the service, making necessary for them to change how the service is invoked. The advantage of our approach is that it centralizes these changes in the adapter class, thus the service consumer does not have to exhaustively look in its application code for every call to the older service. Besides, recent experiments confirm that the additional software layers introduced by the adapter have a negligible impact on applications performance [15, 38].

4. THE EASYSOC ECLIPSE PLUG-IN

Up to now, we have described our catalog of best practices for SOC. To assist developers at practicing the proposed guidelines, we have designed a software tool as a plug-in for the Eclipse IDE, called the EasySOC plug-in. The EasySOC plug-in comprises three modules, each one facilitate the implementation of the set of guidelines explained before. Moreover, the EasySOC plug-in has been employed for empirically validating the benefits of adhering to the proposed guidelines. Sections 4.1 through 4.3 discuss the design and evaluation related to these modules.

4.1. The WSDL Bad Practices Detector

The WSD Bad Practices Detector, or Detector for short, is a module of the EasySOC plug-in that is intended to be used by providers during the service interfaces' specification and whose purpose is automatically checking whether a WSDL document describing the technical contract of a Web Service conforms to the guidelines explained in Section 3.1.

4.1.1. WSDL Bad Practices Detector description The module receives this name since its construction was driven by the catalog of WSDL document discoverability bad practices that we introduced in the study published in [21]. Besides measuring the impact of each bad practice on service discovery, the study assessed the implications of bad practices on developers' ability to make sense of WSDL documents. The catalog consists of eight bad practices and provides a name, a problem description, and a sound refactoring procedure for each one. Although the results of the study motivate bad practices refactoring, manually looking for bad practices in WSDL documents might be a time consuming and complex task. Thus, the Detector consists of eight heuristics to automatically detect each bad practices from the aforementioned catalog.

Since those heuristics are based on the different bad practice definitions, they can be classified according to the type of analysis required to detect the bad practices. Basically, bad practices can be divided into two categories [21]: those that can be detected by analyzing only the structure of WSDL documents, and those whose detection requires a semantic analysis of the names and comments present in WSDL documents.

The heuristics to detect the first kind of bad practices are simple rules based on the commonest bad practice occurrence form. The bad practices that fall into this category are XSD data-types embedded in WSDL documents, which difficult the reuse of the data-types (step 1 in Section 3.1), redundant XML code for defining both data-types and port-types (step 2), and data-types that allow transferring data of any type (also step 2) –or *Whatever types* in EasySOC terminology– which hinders understandability. For example, the rule that detects redundant port-types verifies that a 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. Finally, lacking of comments (step 6) is the only problem related with comments that is supported. Therefore, its heuristic belongs to this type.

As mentioned earlier, detecting the remaining bad practices requires analyzing the semantics of names and comments. Basically, there are three problems that are detected by the associated heuristics: fault information within standard Output messages (step 3), ambiguous names (step 4), and operations of different domains in the same port-type (step 5).

The heuristic for detecting fault information within standard output messages (step 3) warns about the existence of bad practices in this respect when an operation has no Fault message defined, and the comment or some name related with the Output contains one of the following words: *error, fault, fail, exception, overflow, mistake* and *misplay*. This verification is made because these words are commonly related with error conditions when executing a service.

Furthermore, the heuristic related to name ambiguity detection, covers two possible issues. The first is names being too short or too long, for which a rule to check that each name has a length between 3 and 30 characters is provided. The other issue 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 checked by using a probabilistic context free grammar parser [39]. For instance, Figure 5 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.

Finally, the heuristic to determine whether the operations of a service belong to the same domain is based on automatically deducing the domain of each operation exposed by the service, and comparing deduced domains afterward. To do this, the heuristic employs a text classification technique because the only semantic information of an operation a WSDL document provides consists of the names and comments of the operation. To this end, the Rocchio's TF-IDF classifier

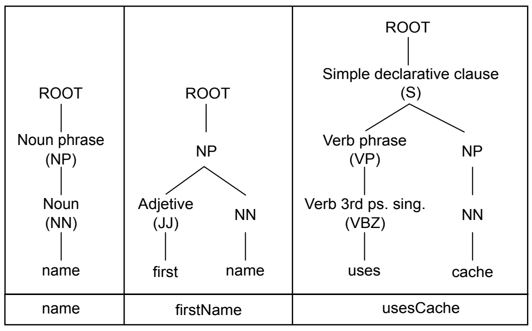


Figure 5. Parsing trees of message part names.

has been selected because empirical studies have showed that it outperforms other classifiers when used in conjunction with Web Services [19]. 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 its name and associated comments as a vector and comparing it to each domain centroid. Finally, the domain associated with the most similar average vector is returned as the domain of that operation.

4.1.2. WSDL Bad Practices Detector evaluation The methodology followed in the evaluation first involved manually analyzing each WSDL document to identify the bad practices 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. Results were organized per bad practice, each bad practice have an associated matrix where the results of both manual and automatic analysis are presented. "Positive" stands for the number of WSLD documents in which the anti-pattern was detected by the analysis, while "Negative" is the number of WSDL documents in which the analysis did not find an anti-pattern occurrence. When the manual classification of a WSDL document is equal to the automatic one, it means that the heuristic accurately operates for that WSDL document. Achieved results are shown in Table II 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 bad practice. The columns of the matrix show the results obtained manually, i.e. the number of WSDL documents that actually had each bad practice.

In the experiments, we used a data-set of 392 WSDL documents [21]. Once each heuristic was fed with and 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 calculated as the number of classifications matching over the total of analyzed WSDL documents. For instance, the accuracy of the *Redundant data model* heuristic was $\frac{221+166}{221+2+3+166} = 98.7\%$. The heuristic for detecting *Low cohesive operations within the same port-type* bad practice achieved the lowest accuracy: 77.5%. One hypothesis that could explain this value relates to potential errors introduced by the classifier, so more experiments are being conducted. Nevertheless, the average accuracy for all the heuristics was 95.8%.

The false positive rate is the proportion of WSDL documents that an heuristic has wrongly labeled as having the corresponding bad practice. In opposition, the false negative rate is the percentage of WSDL documents that an heuristic has wrongly labeled as not having the corresponding bad practice. A false negative rate equals to 1.0 means that a detection heuristic has missed all bad practice occurrences. Therefore, the lower the achieved values the better the detection effectiveness. The average false positive rate was 3.6%, and the average false negative rate was 5.2%, which we believe are encouraging.

Table II. Effectiveness of the Bad Practices Detector. More than 77% per bad practice were automatically detected by this module.

Automatic detection results per bad practice		**Manual detection results** Negative	Positive	**Accuracy**
Enclosed data model	Negative	116	6	98.46%
	Positive	0	270	
Redundant port-types	Negative	161	4	98.98%
	Positive	0	227	
Redundant data models	Negative	221	2	98.72%
	Positive	3	166	
Whatever types	Negative	339	0	99.23%
	Positive	3	50	
Lack of comments	Negative	135	0	100.00%
	Positive	0	257	
Low cohesive operations in the same port-type	Negative	272	10	77.55%
	Positive	78	32	
Ambiguous names	Negative	67	0	99.22%
	Positive	9	316	
Undercover fault information within standard messages	Negative	351	3	98.21%
	Positive	4	34	

Notice that the WSDL Bad Practice detector has some limitation when dealing with bad practices related to natural language, such as inappropriate comments and cohesion. Thus, we are developing different algorithms to increase the detector capabilities. For example, we are working on a heuristic method that uses WordNet to measure how concrete is a comment based on how deep in its associated hyponym tree the comment terms are. In addition, by using WordNet as a taxonomy, the heuristic is able to determine whether a comment is related to its operation signature or not. In general, preliminary results [40] showed that using language corpus combined with sense disambiguation heuristics might effectively determine whether operation comments are meaningless or meaningful.

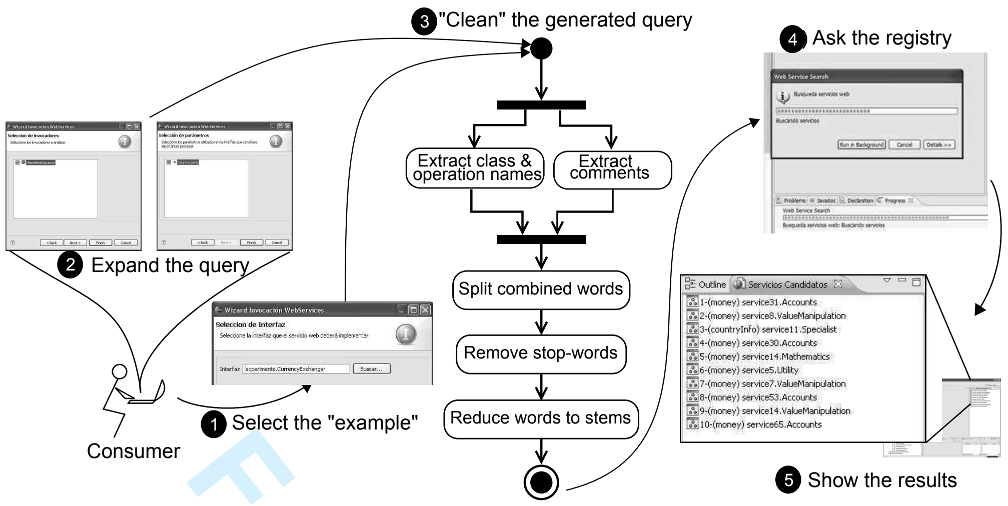
4.2. *The Query Builder*

Another module is intended to help consumers during the development phase of applications, specifically during the service discovery process. In this sense, the EasySOC Query Builder module assists consumers to generate queries, by gathering keywords from the source code of applications.

4.2.1. Query Builder description The module provides a graphical tool (i.e. a wizard) that starts when a consumer selects “Find services for ...” by clicking on an interface that stands for an external service to be outsourced (see stage 1 in Figure 6). Here, the selected interface is the result of employing the first step of the guidelines for making effective queries described.

The wizard uses the Eclipse JDT Search Engine [41] for automatically discovering the internal components that depend on the interface and presents them to the user. The JDT allows automatically discovering the components defined according with the second step of the guidelines.

Similarly, the wizard 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



1-(money) service31.Accounts
2-(money) service8.ValueManipulation
3-(countryInfo) service11.Specialist
4-(money) service30.Accounts
5-(money) service14.Mathematics
6-(money) service5.Utility
7-(money) service7.ValueManipulation
8-(money) service53.Accounts
9-(money) service14.ValueManipulation
10-(money) service65.Accounts

Figure 6. EasySOC stages for generating queries.

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. Then, the user may select or discard the resulting classes (see stage 2 in Figure 6).

The final stage of the query generation process performs a preliminary processing on the classes and the Java interface. As depicted in the center of Figure 6, the module executes a text-mining process for dealing with programming language concerns (see activities "Extract classes & operation names" and "Extract comments"), developers' conventions (activity "Split combined words"), and Natural Language Processing (NLP) related issues (activities "Remove stop-words" and "Reduce words to stems"). All in all, the output of the text-mining process is a list of keywords, whose quantity and quality directly depend on whether consumers followed the third step of the associated guidelines or not.

Although the module allows users to customize queries and test the retrieval effectiveness when using different classes as input, it makes 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.

4.2.2. Query Builder evaluation We evaluated the retrieval effectiveness of the Query Builder by using the previous collection of 392 WSDL documents to feed an information retrieval-based approach to service registry, which is explained [18]. Moreover, undergraduate students from the "Service-Oriented Computing"[‖] course of the Systems Engineering BSc. program (Faculty of Exact Sciences - UNICEN) played the role of service consumers. The students were assigned an exercise consisting on deriving 30 queries based on some test applications, in which each query comprised a Java interface describing the functional capabilities of a potential service. Moreover, the interfaces were designed to be functionally cohesive, and the header and the operations of each interface were commented. For those operations with non-primitive data-types as arguments, their associated 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 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 from combining all these four sources.

[‖]http://www.exa.unicen.edu.ar/~cmateos/cos
[**]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) measures from the information retrieval area. In general, these measures allow assessing the effectiveness of retrieval systems when their results are ranked according to their relevancy [42], and have already been applied in the Web Service discovery area to show how effective different registries are when looking for a service [12].

Basically, Precision-at-n is the percentage of services relevant for a query among the n first returned services. Recall-at-n represent how many of the relevant services are returned in the n first ranked services. R-precision is Precision-at-n when n is the number of relevant services in the registry (R). Finally, NR is a recall metric that also considers the number of registered services, and the rank of the relevant services (r). The Normalized Recall (NR) is one of the most popular measures for evaluating and comparing information retrieval systems because it returns one single number in contrast to paired recall-precision measure [43].

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 calculated each measure 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 measures 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 or not. 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 query generation mechanism very strict. Additionally, it is worth noting that for any query there were, at most, 8 relevant services within the data-set. Besides, there were 10 queries that had associated only one relevant service.

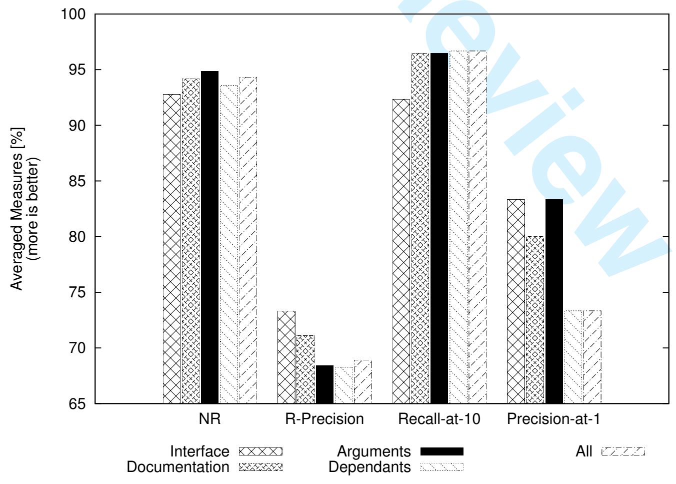


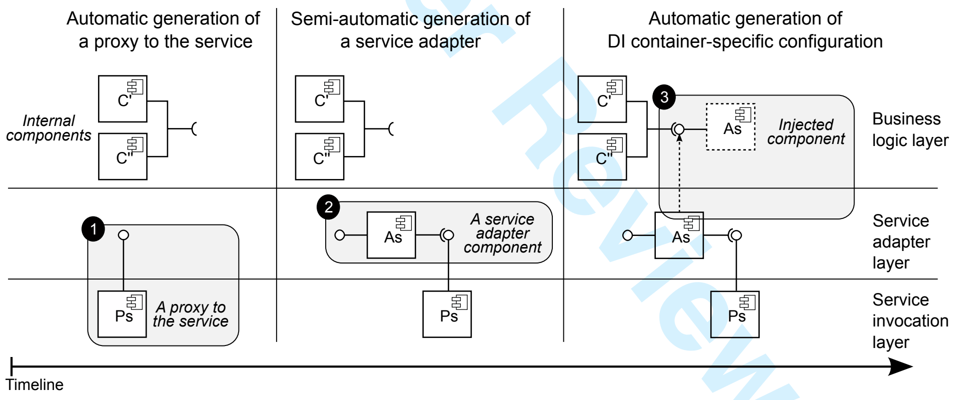
Figure 7. Retrieval effectiveness of the generated queries.

Each bar in Figure 7 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 "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 whereas the possibilities of including it before the 11th positions increased. All in all, for this experiment our Query Builder alleviated discovery by approaching relevant services within a window of 10.

4.3. The Service Adapter

The Service Adapter is the third module and it has been created to assimilate the best practices for applications maintenance. This module goal is to assign service consumers to apply the guideline presented in Section 3.3. To do this, the service adapter module uses as input the interface that the service consumer expects from potential services, and the Web Service that the he/she has selected. Using these inputs, the Service Adapter attempts to automatically write the code needed to adapt the expected interface and the actual service interface. Finally, this module injects the adapter into the service consumer's application, without significant penalties in performance and memory consumption [38].

4.3.1. Service Adapter description The module automatically performs the steps 2 and 3 for the guidelines of Section 3.3 provided step 1 is correctly followed. 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 in an automatic way. Second, the module builds a service 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 service adapters, which is done through Dependency Injection (DI) [37], a popular pattern for seamlessly wiring software components together that is employed by many development frameworks. Figure 8 summarizes the stages that are needed to proxy, adapt, and inject services into applications or another service implementation.



Automatic generation of a proxy to the service
Semi-automatic generation of a service adapter
Automatic generation of DI container-specific configuration
Internal components
C'
C''
1
Ps
A proxy to the service
2
As
A service adapter component
Ps
3
As
Injected component
C'
C''
As
Ps
Business logic layer
Service adapter layer
Service invocation layer
Timeline

Figure 8. EasySOC stages for outsourcing services.

The current implementation of the Service Adapter module uses the Axis2 Web Service library [44] for building service proxies, and Spring [45] 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 [46].

Our algorithm takes two Java interfaces as input and returns the Java code of a service adapter. To do this, it starts by detecting to which operations of one interface should be mapped the operations offered by the other. The algorithm assesses operation similarity by comparing operation names, comments, data-types and names of arguments. 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. To do this, 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, which makes this step semi-automatic.

4.3.2. Service Adapter evaluation 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 the two alternatives for developing a simple, personal agenda by outsourcing services from the above data-set. We made sure that several services offering similar functionality but exposed by different providers existed in the data-set. It is worth mentioning that our goal was not to assess the accuracy of our mapping algorithm, but the impact of the application design that results from employing the guidelines in maintainability.

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 proposed guidelines with respect to the traditional approach. We employed the well-known SLOC (Source Lines Of Code), Ce (Efferent Coupling), CBO (Coupling Between Objects) and RFC (Response For Class) software engineering metrics.

In general, these metrics allow assessing maintainability concerns on source code. In particular, SLOC counts the total non-commented and non-blank lines across the entire application code[††], including the code implementing the pure application logic, plus the code for interacting with the various Web Services. The smaller the SLOC value is, the less the amount of source code that is necessary to maintain once an application has been implemented.

Ce indicates how much the classes and interfaces within a package depend upon classes and interfaces from other packages [47]. In other words, this metric includes all the object types within the source code of the target package referring to the object types not in the target package. In our case, as the proxy code does not depend upon the code implementing the application logic, Ce will just refer to the number of efferent couplings of the classes/interfaces that depend upon proxy classes/interfaces. Under this condition, the less the Ce value is, the less the dependency between the functional code of an application and the interfaces representing server-side service contracts. The utility of Ce in our evaluation is for determining what the influence of the adapter layer of EasySOC on this kind of dependency is.

CBO is the amount of classes to which an individual class is coupled [48]. For example, if a class A is coupled to two more classes B and C, its CBO is two. In this sense, the less a class is coupled to other classes, the more the chance of reusing it. Since reusability is one of the components of maintainability [49], CBO can be used as a complementary indicator of how maintainable a software is.

Finally, RFC counts the number of different methods that can be potentially executed when an object of a target class receives a message, including methods in the inheritance hierarchy of the class as well as methods that can be invoked on other objects [48]. Note that if a large number of methods are invoked in response to receiving a message, testing becomes more difficult since a greater level of understanding of the code is required. Since testability is also one of the components of maintainability [49], it is highly desirable to achieve low RFC values for application classes.

Table III shows the resulting metrics values for the four implementations of the personal agenda: traditional, guidelines-based, 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.

[††]As defined in the COCOMO cost estimation model

Table III. Personal agenda: Source code metrics.

Variant		Id	SLOC	Ce	CBO	RFC
Initial Web Service providers	Traditional	C_1	242	7	4.50	30.00
	Guidelines-based	E_1	309	7	1.70	7.20
Alternative Web Service providers	Traditional	C_2	246	10	4.67	22.67
	Guidelines-based	E_2	327	10	2.00	7.45

From Table III, it can be seen that the variants using the same set of service providers resulted in equivalent Ce values: 7 for C_1 and E_1, and 10 for C_2 and E_2. This means that the variants generated via the proposed guidelines (E_x), did not incur in extra efferent couplings with respect to the traditional variants (C_x). Moreover, if we do not consider the corresponding service adapters, Ce for the variants drops down to zero because following the guidelines of Section 3.3 effectively pushes the source 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 helps in simplifying service replacement.

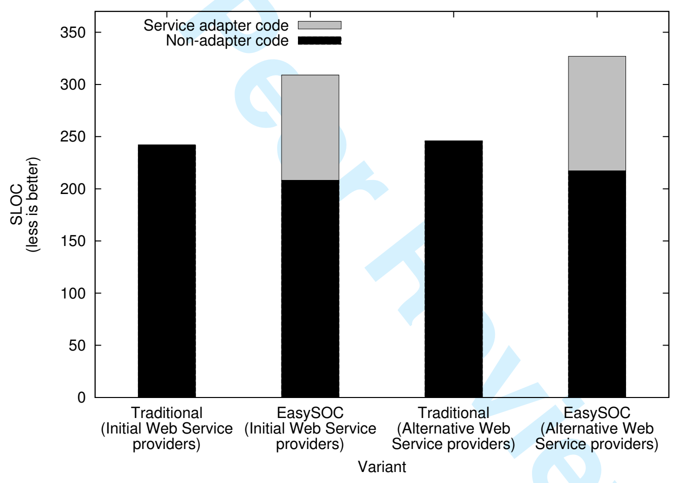


Figure 9. Source Lines of Code (SLOC) of the different applications.

Figure 9 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. Nevertheless, 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 C_1 (more than 7% of its code).

The variants coded following the guidelines 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, and had only business logic code because, unlike its traditional counterpart, it did not include statements for importing and instantiating proxy classes and handling Web Service-specific exceptions. There is another positive aspect concerning service adapters and SLOC. Basically, 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 C_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 tested application.

CBO and RFC metrics were also computed (Figure 10 and Figure 11, respectively). 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 representing service descriptions, should be kept to a minimum. 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 guidelines-based and the traditional approach exhibited increased values when changing the provider for a Web Service. RFC, on the other hand, presented a less uniform behavior.

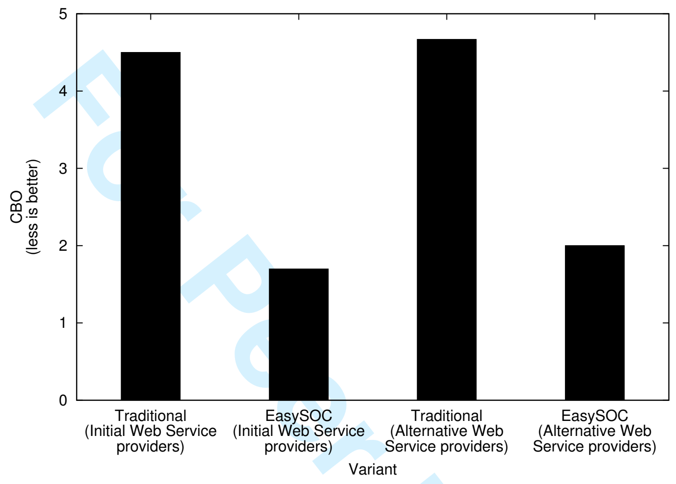


Figure 10. Coupling Between Objects (CBO) of the different applications. The lower values achieved by guidelines-compliant versions imply that they will be easier to maintain.

As quantified by Ce, following the guidelines did not reduce the amount of efferent couplings from the package implementing the application logic. Naturally, the reason for this is that the service descriptions adhered by E_x are the same as C_x. However, the guidelines-based 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. service 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 11, this separation also helped in achieving better average RFC. Moreover, although the plain sum of the RFC values of the applications when following the guidelines -E_x- were greater compared to C_x, the total RFC of the classes implementing application logic (i.e. without taking into account service adapter classes) were both smaller. This suggests that the pure application logic of E_1 and E_2 is easier to understand than C_1 and C_2. Arguably, in large projects, much of the source code of guidelines-based applications will be application logic instead of service adapters. Hence, preserving the understandability of this kind of code is crucial.

Finally, implementing service adapters (see Section 3.3) is at present the Achille's heel of our EasySOC supporting tool, as the task of building service adapters is not trivial and requires to recompile the system to change the service. We will investigate heuristics to automatically match service consumer interfaces onto actual service contracts. We aim at generating adapter code that solves ambiguities in signatures by comparing for example operation names and data-type structures of input/output parameters. Our goal is similar to that of efforts like , but we aim at performing adaptations at the consumer application side. Fourth, we plan to develop guidelines for consuming and providing semantically annotated services. Regarding consumption,

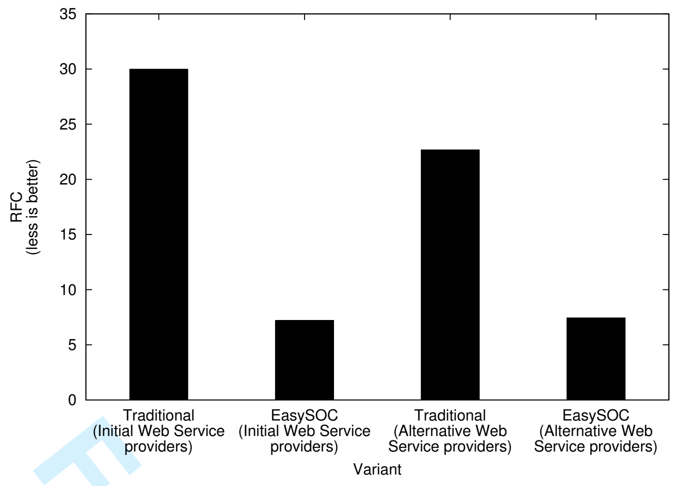


Figure 11. Response For Class (RFC) of each application variant. The lower the achieved values the easier the maintenance. Variants resulting from following our guidelines achieved the lowest values.

we have designed a programming model for developing semantic Web Services-aware applications. This model facilitates the management of semantic annotations when developers follow classic Java Beans programming conventions. Currently, we are gathering preliminary results related to developers' acceptance of the model and best practices for adopting it. At the same time, we are in the process of conceptualizing lessons learned from employing the approach to annotate Web Services with semantic data described in [50]. Last but not least, we are planning to develop EasySOC supporting tools for other widely used platforms (e.g. .NET) and IDEs in order to bring our ideas to a broader development community.

5. CONCLUSIONS

The software industry is embracing the Service-Oriented Computing (SOC) paradigm as the premier approach for building reusable services and applications in heterogeneous, distributed computing environments. However, SOC presents many intrinsic challenges that both Web Service providers and consumers must face.

Historically, catalogs of best practices have been widely recognized as a very valuable and helpful mean to software practitioners for dealing with common problems in many different contexts. In this sense, this paper presented a catalog of concrete, accessible, validated guidelines for avoiding recurrent problems [19, 21, 20] when designing and implementing Web Services and applications.

The practical implications of our guidelines have been corroborated experimentally, which suggests that our work can be conceived as being best practices to be readily employed in the software industry. In particular, we have assessed the impact of our guidelines for improving Web Service descriptions by employing three registries simultaneously supporting service discovery and human consumers, who have the final word on which service is more appropriate. Results showed that improved descriptions are easier to understand than their "raw" counterpart [21]. Similarly, the positive effect on service discovery of the guidelines for generating and expanding queries has been also evidenced [18]. Also, the implications of the guidelines for consuming services on Web Service and application maintenance have been formally and experimentally corroborated in [38] and [19] respectively. It is worth mentioning that a step towards experimentally assessing the effects of simultaneously following our guidelines can be found in [15], in which we described the results of combining both the guidelines to generate effective queries and shield applications from specific services.

In addition, the guidelines for defining easily understandable have been applied in a real-life development [8]. This development consisted in migrating a COBOL legacy system to a SOC system. Actually, this work presents a case study in which developers did not take into account the WSDL document quality. As a result of this, the development of service consumer applications was too complex and error prone. Hence, the migration had to be re-done for improving the system WSDL documents proving the importance of good quality WSDL document.

Clearly, building loose coupled applications using the corresponding guidelines, imposes a radical shift in the way applications are developed by the software industry. This means that a company willing to employ the proposed guidelines to start producing applications would have to invest much time in training its development team, which results in a costly start-up curve. Recently, the impact of the proposed guidelines and its supporting tool-box on the software development process itself from an engineering point of view has been partially assessed in [20]. Concretely, we performed further experiments to test the following hypothesis: understanding common 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 applications in accordance with the proposed guidelines, which should sharpen the learning curve needed to develop loose coupled service-oriented applications. Results showed that they perceived that the proposed approach is convenient and thus may be easily adopted. In the near future, we will conduct experiments with other students and real development teams to further validate our claims in this respect.

Furthermore, our work will be extended in several directions. First, the common way of generating service contracts in the industry is by deriving them from the corresponding implementation code. This approach is called code-first because contracts are obtained *after* a developer implements a functionality that is to be exposed as a service. Although the guidelines for improving service descriptions have proved to be effective for building well-defined contracts, revising an already generated WSDL impacts on the associated implementation. Therefore, we will investigate and elaborate a set of early indicators to help developers avoiding WSDL anti-patterns during coding time.

Finally, since REST services are a current trend to develop some SOA systems [51], we are working on identifying good practices when using REST services. This represents in itself a very interesting research opportunity but at the same time potentially requires rethinking or adapting some of our guidelines. Furthermore, we are studying the benefits of consuming REST services versus traditional Web Services on mobile devices. Our analysis is focused on reducing not only battery consumption, but also network usage, which might impose important cost on final users. As a case study, we will employ some REST Web Services from a real Internet-based agricultural system. We are at present implementing client applications for calling these services that target Android-based devices, from which we perform the measurements.

ACKNOWLEDGMENTS

We acknowledge the financial support provided by ANPCyT through grant PAE-PICT 2007-02311.

REFERENCES

[1] Bichler M, Lin KJ. Service-Oriented Computing. *Computer* 2006; **39**(3):99–101.

[2] Erickson J, Siau K. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of Database Management* 2008; **19**(3):42–54.

[3] Atkinson M, DeRoure D, Dunlop A, Fox G, Henderson P, Hey T, Paton N, Newhouse S, Parastatidis S, Trefethen A, *et al.*. Web Service Grids: An evolutionary approach. *Concurrency and Computation: Practice and Experience* 2005; **17**(2-4):377–389.

[4] Campbell-Kelly M. The rise, fall, and resurrection of software as a service. *Communications of the ACM* 2009; **52**(5):28–30, doi:http://doi.acm.org/10.1145/1506409.1506419.

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

[6] W3C Consortium. XML Schema Definition Language (XSD) 1.1 part 1: Structures. W3C Working Draft, http://www.w3.org/TR/xmlschema11-1 2009.

[7] Loughran S, Smith E. Rethinking the java soap stack. *IEEE International Conference on Web Services*, 2005.

[8] Rodriguez J, Crasso M, Mateos C, Zunino A, Campo M. Bottom-up and top-down cobol system migration to web services: An experience report. *Internet Computing, IEEE* 2011; **PP**(99):1, doi:10.1109/MIC.2011.162.

[9] Peiris C, Mulder D, Cicoria S, Bahree A, Pathak N. Introducing service-oriented architecture. *Pro WCF*. Apress, 2007; 3–24. URL http://dx.doi.org/10.1007/978-1-4302-0324-7_1.

[10] OASIS Consortium. UDDI version 3.0.2. UDDI Spec Technical Committee Draft, http://uddi.org/pubs/uddi_v3.htm Oct 2004.

[11] Garofalakis J, Panagis Y, Sakkopoulos E, Tsakalidis A. Contemporary Web Service Discovery Mechanisms. *Journal of Web Engineering* 2006; **5**(3):265–290.

[12] Crasso M, Zunino A, Campo M. A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management* 2011; **22**:103–134.

[13] Kousiouris G, Kyriazis D, Varvarigou T, Oliveros E, Mandic P. *Achieving Real-Time in Distributed Computing: From Grids to Clouds*, chap. Taxonomy and State of the Art of Service Discovery Mechanisms and Their Relation to the Cloud Computing Stack. IGI Global, 2012; 75–93.

[14] McCool R. Rethinking the Semantic Web, part II. *IEEE Internet Computing* 2006; **10**(1):96, 93–95, doi:10.1109/MIC.2006.18.

[15] Crasso M, Mateos C, Zunino A, Campo M. Empirically assessing the impact of dependency injection on the development of Web Service applications. *Journal of Web Engineering* 2010; **9**(1):66–94. URL http://www.rintonpress.com/journals/jweonline.html#v9n1.

[16] Erl T, Karmarkar A, Walmsley P, Haas H, Yalcinalp LU, Liu K, Orchard D, Tost A, Pasley J. *Web Service Contract Design and Versioning for SOA*. 1 edn., Prentice Hall PTR: Upper Saddle River, NJ, USA, 2009.

[17] Crasso M, Rodriguez JM, Zunino A, Campo M. Revising WSDL documents: Why and how. *IEEE Internet Computing* 2010; **14**(5):30–38, doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2010.81.

[18] Crasso M, Zunino A, Campo M. Combining query-by-example and query expansion for simplifying Web Service discovery. *Information Systems Frontiers* 2011; **13**:407–428. URL http://dx.doi.org/10.1007/s10796-009-9221-9.

[19] Crasso M, Mateos C, Zunino A, Campo M. EasySOC: Making Web Service outsourcing easier. *Information Sciences* 2010; URL http://dx.doi.org/10.1016/j.ins.2010.01.013, to appear.

[20] Mateos C, Crasso M, Zunino A, Campo M. A software support to initiate systems engineering students in Service-Oriented Computing. *Computer Applications in Engineering Education* 2011; To appear.

[21] Rodriguez JM, Crasso M, Zunino A, Campo M. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming* 2010; **75**(11):1001–1021, doi: http://dx.doi.org/10.1016/j.scico.2010.01.002.

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

[23] Kohlborn T, Korthaus A, Chan T, Rosemann M. Identification and analysis of business and software services - a consolidated approach. *IEEE Transactions on Services Computing* 2009; **2**(1):50–64, doi:http://dx.doi.org/10.1109/TSC.2009.6.

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

[25] Flaxer D, Nigam A. Realizing business components, business operations and business services. *IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-EAST'04)*, IEEE Computer Society, 2004; 328–332, doi:http://dx.doi.org/10.1109/CEC-EAST.2004.55.

[26] Papazoglou M, van den Heuvel WJ. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology* 2006; **2**(4):412–442, doi:http://dx.doi.org/10.1504/IJWET.2006.010423.

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

[28] Blake MB, Nowlan MF. Taming Web Services from the wild. *IEEE Internet Computing* 2008; **12**(5):62–69, doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2008.112.

[29] Allen EE, Cartwright R. Safe instantiation in generic java. *Science of Computer Programming* 2006; **59**(1-2):26 – 37, doi:DOI:10.1016/j.scico.2005.07.003. Special Issue on Principles and Practices of Programming in Java (PPPJ 2004).

[30] Pasley J. Avoid XML schema wildcards for Web Service interfaces. *Internet Computing, IEEE* May-June 2006; **10**(3):72–79, doi:10.1109/MIC.2006.45.

[31] Yourdon E, Constantine LL. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1979.

[32] Bieman J, Ott L. Measuring functional cohesion. *Software Engineering, IEEE Transactions on* aug 1994; **20**(8):644 –657, doi:10.1109/32.310673.

[33] Basili V, Briand L, Melo W. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on* oct 1996; **22**(10):751 –761, doi: 10.1109/32.544352.

[34] Juric MB, Sasa A, Brumen B, Rozman I. WSDL and UDDI extensions for version support in web services. *Journal of Systems and Software* 2009; **82**(8):1326–1343. SI: Architectural Decisions and Rationale.

[35] Duftler M, Mukhi N, Slominski A, Weerawarana S. Web Services Invocation Framework (WSIF). *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, ACM Press, 2001.

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

[37] Johnson R. J2EE development frameworks. *Computer* 2005; **38**(1):107–110.

[38] Mateos C, Crasso M, Zunino A, Campo M. Separation of concerns in service-oriented applications based on pervasive design patterns. *Web Technology Track (WT) - 25th ACM Symposium on Applied Computing (SAC '10)*, ACM Press, 2010; 2509–2513. URL http://doi.acm.org/10.1145/1774088.1774263.

[39] Klein D, Manning C. Accurate unlexicalized parsing. *41st Annual Meeting on Association for Computational Linguistics (ACL'03)*, Association for Computational Linguistics, 2003; 423–430, doi:http://dx.doi.org/10.3115/1075096.1075150.

[40] Rodriguez J, Crasso M, Zunino A, Campo M. Automatically detecting opportunities for web service descriptions improvement. *Software Services for e-World*, *IFIP Advances in Information and Communication Technology*, vol. 341, Cellary W, Estevez E (eds.), Springer Boston, 2010; 139–150.

[41] The Eclipse Foundation. Eclipse Java development tools (JDT). http://www.eclipse.org/jdt 2010.

[42] Korfhage RR. *Information Storage and Retrieval*. John Wiley & Sons, Inc.: New York, NY, USA, 1997.

[43] Bollmann P. The normalized recall and related measures. *Proceedings of the 6th annual international ACM SIGIR conference on Research and development in information retrieval*, 1983; 122–128.

[44] Perera S, Herath C, Ekanayake J, Chinthaka E, Ranabahu A, Jayasinghe D, Weerawarana S, Daniels G. Axis2, middleware for next generation Web Services. *IEEE International Conference on Web Services (ICWS'06)*, IEEE Computer Society, 2006; 833–840, doi:http://doi.ieeecomputersociety.org/10.1109/ICWS.2006.36.

[45] Walls C, Breidenbach R. *Spring in Action*. Manning, 2005.

[46] Stroulia E, Wang Y. Structural and semantic matching for assessing Web Service similarity. *International Journal of Cooperative Information Systems* 2005; **14**(4):407–438.

[47] Martin RC. Object-Oriented Design Quality Metrics: An Analysis of Dependencies. *Report on Object Analysis and Design* 1995; **2**(3).

[48] Chidamber SR, Kemerer CF. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.

[49] International Organization for Standardization. Software engineering - product quality - part 1: Quality model. *ISO 9126* 2001; .

[50] Crasso M, Zunino A, Campo M. Combining document classification and ontology alignment for semantically enriching Web Services. *New Generation Computing* 2010; **28**:371–403. URL http://www.ohmsha.co.jp/ngc/abstract/28-4-3.htm.

[51] Pautasso C, Zimmermann O, Leymann F. RESTful Web Services vs. "big" Web Services: making the right architectural decision. *Proceeding of the 17th international conference on World Wide Web*, WWW '08, 2008; 805–814.