EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

EasySOC: Making Web Service Outsourcing Easier

Marco Crasso, Cristian Mateos*, Alejandro Zunino*, Marcelo Campo*

ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682 ext. 35. Fax.: +54 (2293) 439683
Also Consejo Nacional de Investigaciones Cientficas y Tcnicas (CONICET)

Abstract

Service-Oriented Computing has been widely recognized as a revolutionary paradigm for software development. Despite the important benefits this paradigm provides, current approaches for service-enabling applications still lead to high costs for outsourcing services with regard to two phases of the software life cycle. During the implementation phase, developers have to invest much effort into manually discovering services and then providing code to invoke them. Mostly, the outcome of the second task is software containing service-aware code, therefore it is more difficult to modify and to test during the maintenance phase. This paper describes EasySOC, an approach that aims to decrease the costs of creating and maintaining service-oriented applications. EasySOC combines text mining, machine learning, and best practices from component-based software development to allow developers to quickly discover and non-invasively invoke services. We evaluated the performance of the EasySOC discovery mechanism using 391 services. In addition, through a case study, we conducted a comparative analysis of the software technical quality achieved by employing EasySOC versus not using it.

Keywords: service-oriented computing, service outsourcing, text mining, machine learning, dependency injection

1. Introduction

Service-Oriented Computing (SOC) [20] is a new computing paradigm that supports the development of distributed applications in heterogeneous environments. With SOC, distributed systems are built by assembling together existing functionalities, or *services*, that are published in a network. A service is a piece of software that is wrapped with a network-addressable interface, which exposes its capabilities to the outer world. From a software engineering standpoint, SOC is an interesting paradigm since it heavily promotes software reuse in a loosely coupled way [20].

Mostly, the software industry has adopted SOC by using Web Service technologies. A Web Service is a program with a well-defined interface that can be located, published, and invoked by using ubiquitous Web protocols [55, 11]. Basically, the Web Service model encompasses three elements: service providers, service requesters, and service registries. Service providers use an XML-based language called WSDL [58] to create documents describing their Web Services, and publish these documents in registries, a.k.a. UDDI registries [38]. Service requesters can use the registry to find a Web Service that matches their needs, and then invoke its operations by using the corresponding WSDL document. WSDL and UDDI are standards designed to set the

*Corresponding author.
Email address: cmateos@exa.unicen.edu.ar (Cristian Mateos)

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

basis for interoperability among clients and services in environments where many technologies can be found.

Despite the important benefits Web Services provide, namely loose coupling among service consumers and providers, and high levels of global interoperability, Web Service technologies are currently not broadly used [35, 59]. Roughly, the cause of this fact is that current approaches to service consumption from within applications require developers to manually look for suitable services and "glue" them in their client-side code afterward. This not only forces developers to invest burdensome efforts into discovering services and providing code to invoke the selected ones, but also leads to software containing service-aware code. We refer as service-aware code to those parts of a client application that are tightly coupled to the interface provided by specific providers. In an open world setting, where services are built by different organizations, it is not necessarily true that all the available implementations of an abstract service have the same interface [5]. Therefore, changing service providers requires changing the application logic as well. Thus, service-aware code is more difficult to modify and test. Then, the tasks of developing and maintaining a SOC application become hard.

The problem associated with the development of service-oriented applications may stem from the fact that discovering services that fulfill the *functional* expectations of the client through common service registries is "as finding a needle in a haystack" [17] when the number of services is large, which is the case of massively distributed environments like the Web. The problem associated with the maintainability of such applications is a consequence of the approach commonly used by developers to invoke a Web Service, which consists in obtaining the WSDL document of the service, interpreting it, and generating a client-side proxy to the remote service. Though this approach allows designers to separate business logic from the code for invoking services, the application logic mixes up with code that is subordinated to particular service interfaces. This fact reduces the internal quality of the resulting software, in which modifiability and out-of-the-box testing (i.e. outside a SOC setting) are compromised. In particular, having good maintainability is essential, because software maintenance costs represent around 50% of the total software life-cycle cost [24].

We claim that it is necessary to further simplify the process of service-oriented software development and maintenance. First, discovering and selecting existing services must not be a tedious and time-consuming task for developers. Second, invoking services should be as non-intrusive to the application logic as possible, thus diminishing the effort of modifying and testing the client-side functionality once it has been implemented. This paper proposes EasySOC, an approach for making the task of outsourcing functionality in service-oriented software easier, which essentially provides means for efficiently discovering third-party services, and enforcing minimum source code provision in the application logic for consuming them.

EasySOC promotes separation of concerns between the application logic and the functionality related to service engagement. The approach lets developers to focus on implementing and testing the functional code of an application, and then "SOC--enable" it by discovering and loosely assembling the external functionality. To this end, EasySOC requires designers to specify the potential Java interface of the services to outsource. Then, EasySOC uses text mining techniques for automatically pulling out relevant information about the desired service from the source code of the client-side software. EasySOC uses a Query-by-Example (QBE) approach to look for relevant third-party services based on this information, i.e. the example, which is supported by a search space reduction mechanism that uses machine learning techniques to allow discoverers to promptly select a service from a wieldy list of candidates. In this

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

sense, EasySOC aims to make Web Service candidate selection easier for humans, i.e. automatic service selection is not addressed here.

After discovery, the selected services are non-invasively integrated with the application by using the Dependency Injection (DI) [23] design pattern. With DI, external services are injected into application components requiring these services without affecting the components' implementation. Furthermore, we combine DI with the Adapter Design Pattern to establish loose relationships between clients and service interfaces of specific providers. In this respect, EasySOC does not represent a new programming paradigm for SOC but an approach that exploits DI to build more maintainable service-oriented applications.

The contribution of this work is a development model for building maintainable SOC applications. At the heart of this model is a semi-automatic service outsourcing process that allows developers to quickly find and non-invasively consume Web Services. Moreover, experimental results show that when using the information of the code of EasySOC-based applications to generate queries, our service search engine was more effective not only in retrieving more relevant services within a window of 10 candidates but also in ranking them first in the result list, compared with the discovery performance resulted from generating queries from non-EasySOC code [10].

The rest of the paper is organized as follows. The next section discusses the most relevant related work. Section 3 takes a deeper look at the EasySOC approach. Section 4 presents a detailed evaluation of the approach. Section 5 concludes the paper.

2. Related work

As suggested in the previous paragraphs, EasySOC represents a new development model for SOC applications. The model is based on an iterative approach to service outsourcing, where each iteration comprises three steps: (1) finding the list of candidate Web Services for the particular i^{th} service being outsourced, (2) select a candidate service from the resulting list, and (3) invoking the selected service from within the client-side application. Steps (1) and (3) are automatically performed by EasySOC via text mining and machine learning techniques, and DI, respectively, whereas step (2) is manually carried out by the developer.

In this Section we position related work against the *automatic* steps of the EasySOC outsourcing model, namely step (1) or Web Service discovery (next Subsection) and step (3) or Web Service consumption (Subsection 2.2).

2.1. Approaches to Web Service discovery

Recently, the problem of finding proper services has been receiving much attention from both the academia and the industry. [17] presents a comprehensive survey of methods, architectures and models for discovering Web Services that discusses over 30 proposals. Broadly, some of these efforts propose to combine Web Services and Semantic Web technologies [49], whereas others aim to take advantage of classic Information Retrieval (IR) techniques. Within the former group, some approaches [15, 36] define a meta-ontology for modeling Web Services, which allows publishers to associate concepts from shared ontologies with services. Similarly, WSDL-S [51] is an attempt to extend WSDL with semantic capabilities. This enables the use of semantic matching algorithms to very effectively find required services. Furthermore, by exploiting unambiguous service definitions and semantic matching, software agents can automate the process of finding, invoking, and composing Web Services [39, 34].

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

However, building ontologies is a costly and error-prone task [18, 50], and there is a lack of both widely-adopted standards for representing ontologies and publicly available Semantic Web Services [35]. Besides, using ontologies forces publishers and discoverers to be proficient in semantic technologies, and imposes modifications on the current, syntactic UDDI infrastructure [4].

With respect to IR-inspired service discovery, [13, 53] adapt the Vector Space (VS) model for representing textual information available in Web Service descriptions and queries as vectors, then service look up operates by comparing such vectors. Concretely, the vector representing a query is matched against the vectors within the VS (i.e. the available services). The service whose vector maximizes the spatial nearness to the query vector is retrieved. Here, the number of matching operations is proportional to the number of published services. Thus, despite being suitable for Intranet settings, where the number of available services is usually small, this approach may have performance problems in distributed environments, such as WANs or the Internet, where the number of services is large, making it unsuitable for agilely responding to user requests. Another shortcoming of IR-based approaches is that their effectiveness depends on how explanatory the words included in queries and service descriptions are, because these words represent vector elements within the VS. In other words, on one hand it depends on publishers' use of best practices for naming and documenting services, and discoverers' ability to describe what they are looking for, on the other hand. Assuming that developers tend to follow best practices for naming and documenting services, so that services and their descriptions can be understood and re-used by other developers, the descriptiveness of queries has recently received attention from academia for its potential effects on discovery.

Deriving queries to find Web Services from design-time specifications is explored in [29]. Under this approach, service-oriented applications are designed with the help of certain models that extend the UML notation. These extended models allow designers to indicate, using a very expressive query language, whether an individual class operation will be implemented in-house or delegated to a third-party service. Moreover, designers can specify constraints on the services/operations that will be outsourced (e.g. provider, the number of parameters of an operation, etc.). To compute the similarity between a query and the available services, a two-step process is used. Firstly, the services satisfying the specified constraints are retrieved. Secondly, the service operations that best match the query are determined through a similarity heuristic that is based on graph-matching techniques. The approach has, however, some drawbacks. On one hand, application designers have to learn and adopt the extended UML notation and the query language, and queries may be rather hard to define. On the other hand, designs of existent service-oriented applications must be adapted to this new notation so as to enable service discovery. In contrast, EasySOC derives those queries directly from existing application code, i.e. EasySOC uses the information already present in the interfaces describing outsourced services and the context in which these interfaces are reached. This allows developers to *implicitly* state queries by using nothing but their preferred programming language.

Lastly, the idea of extracting information from the client application and using it for creating service queries has been also promoted by SAGE [1]. SAGE proposes to employ a personal software agent for assisting a developer in finding Web Services based on the knowledge of the development environment (e.g. an IDE). Basically, this agent periodically monitors the developer until it detects an action that may be associated with requesting a service. The agent then uses any captured textual input and certain contextual information (e.g. the name of the project the user is working on, the developer's role, etc.) to search service repositories in background. When a relevant service is discovered, the agent presents the results to the user, who must decide what to do with the service (options are to execute it, not to execute it, or defer the decision). In this way, the agent gradually infers the user's preferences with regard to whether a retrieved Web Service should be used or not. The uttermost goal of SAGE is to automatically execute or discard services in new and similar situations.

2.2. Approaches to Web Service consumption

To address the problem of easily invoking Web Services from within applications, some toolkits (e.g. JWSDP[1]) and frameworks (e.g. WSIF and CXF) have been built. Basically, they provide programming abstractions to keep the application code as clean as possible from Web Service implementation details. These solutions follow a contract-first approach to service consumption. We refer as contract-first approach to those approaches that first obtain the interface, or contract, of the outsourced service, and create/modify the application components that use it afterward. A contract establishes the terms of engagement of an individual service, providing technical constraints and requirements (e.g. specific data-types) as well as any information the provider of the service makes public [14]. Thus, the application logic is inevitably dependent on specific service contracts. This makes application testing, modifiability and adaptability difficult. A more flexible solution to these issues is achieved by the Dynamic Proxy Invocation (DPI) approach. This approach associates client-side code with abstract service descriptions. Then, at runtime, a Web Service whose interface exactly adheres to the abstract description is retrieved and integrated with the application through a proxy. Although DPI allows developers to effortlessly swap over different services that provide the same interface, services whose interfaces are somewhat dissimilar to the abstract description but they deliver the required functionality cannot be easily integrated.

Web Services Management Layer (WSML) [7] specifically addresses the problem of non-invasively integrating Web Services with applications. Conceptually, WSML introduces a software layer that isolates applications from concrete service providers. Within this layer, a special component or proxy is responsible for representing a set of "semantically" similar Web Services yet potentially exposing different interfaces. In other words, the proxy hides the syntactical differences among services providing the same functionality. Applications invoke services through these proxies, which intercept, adapt and forward individual requests to concrete Web Services based on user-provided adapters coded in JAsCo [54]. JAsCo is an AOP language that supports dynamic deployment of new adapters. A limitation of WSML is that developers have to learn not only a new programming language but also new programming abstractions, because even when the syntax of JAsCo is similar to that of Java, its semantics are quite different. Besides, although the authors in [7] have meticulously discussed WSML, the soundness of the approach has not been corroborated experimentally. Finally, WSML provides an extensible support for proxies to tune service access. For example, a proxy associated with *N* different service providers may be configured to use the provider that historically has offered the best response time. A limitation of this mechanism is that, initially, providers have to be manually discovered.

Similar to [7], [45] uses AOP to dynamically integrate Web Services with applications. The implementation of any internal method can be replaced by a Web Service

[1] Java Web Services Development Pack `http://java.sun.com/webservices/jwsdp/index.jsp`

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

operation by declaring an aspect that intercepts the execution of that method. The aspect receives the WSDL document of the service, through glue code implemented by the developer, and executes operations on the Web Service. Aspects are implemented in AspectJ [25], a language that extends Java with AOP constructs. [45] includes a service discovery system that allows developers to find services by specifying their potential inputs and outputs. Then, when a relevant service is found, aspect code is generated and deployed to invoke the corresponding Web Service. Queries have the same structure as the *message* element of the WSDL language, which is used to describe service inputs/outputs in the XSD (XML Schema Definition) language. Therefore, building queries also requires developers to specify the expected data-types for service operations in XSD, which is a tedious task [9]. Finally, [45] aims at fully automating the tasks of discovery and integration of services at runtime, which have received some criticism [42]. In real world scenarios, some characteristics of the Web Service engagement process, such as the need for establishing service-level agreements, performing payment or determining the provider's reputation still clearly requires an active intervention from the user.

To conclude, [37] presents a semi-automated approach to generate service representatives that are similar to EasySOC *service adapters*, which result from combining DI and the Adapter design patterns. Essentially, the approach identifies structural differences between two service interfaces, such as parameter types, missing/extra parameters and parameter ordering, and builds a *mismatch tree*. Then, for the mismatches that can be resolved automatically, adapter code is generated. The mismatches that require developers' input for their resolution are conveniently presented to the user through a GUI. Note that this ideas may be also applied to further ease the implementation of EasySOC service adapters.

EasySOC copes with the mentioned shortcomings. Firstly, since EasySOC discovery technique is based on the VS approach, it proposes a search space reduction mechanism that greatly mitigates the inability of such approaches to handle large data-set in interactive usage scenarios, this is, those in which only the user can perform candidate service selection. In addition, by automatically inferring potential service descriptions from the information present in client-source code, EasySOC frees developers from generating queries. Secondly, our approach is based upon a DI-inspired programming model that shields application logic from not only service invocation details but also providers' contracts. As a consequence, switching between available providers for an outsourced functionality is easier and cheaper –with regard to software modifiability and maintainability– than contract-first or DIP-based alternatives. Moreover, the code to perform contract adaptation is specified in the same programming language as the pure functional code, that is, there is no need to learn any new language or programming paradigm.

3. The EasySOC approach

Component-based software development is a branch of software engineering that focuses on building software in which functionality is split into a number of logical software components with well-defined interfaces. Components are designed to hide their associated implementation, to not share state, and to communicate with other components via message exchanging. Anatomically, a component can be thought as an object from the object-oriented (OO) paradigm, and the interface(s) to which the object adheres. The spirit of the component-based paradigm is that application compo-

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

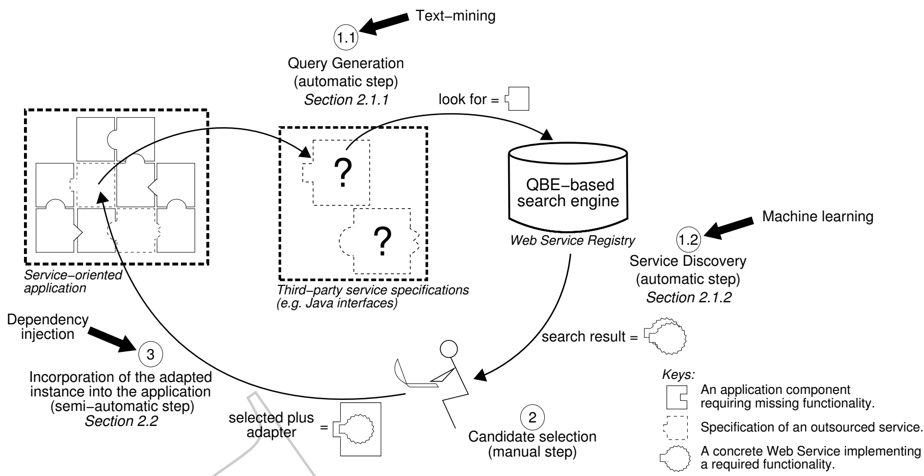


Figure 1: Overview of EasySOC

nents only know each other’s interfaces, thus high levels of flexibility and reuse can be achieved.

SOC has evolved from component-based notions to face the challenges of software development in heterogeneous distributed environments [40], where interoperability is a crucial issue not yet fully addressed, nevertheless it suggests unprecedented levels of reusability. A service-oriented application can be viewed as a component-based application that is created by assembling two types of components: *internal*, which are those locally embedded into the application, and *external*, which are those statically or dynamically bound to a service. When building a new application, a software designer may decide to provide an implementation for some application component, or to reuse an existing implementation instead. From now on, we will refer to this latter as *outsourcing*. In this context, to outsource a component C means to fill the hole left by the missing functionality with the one implemented by an existing service S. As there may be many published services that serve to this purpose, an early problem is how to allow developers to effectively and quickly discover candidate services. After discovering, a latter problem is how to allow developers to integrate outsourced services with their software while achieving good maintainability. Note that addressing these problems would minimize the impact of outsourcing on the software life cycle, in particular on development and maintenance.

To address these problems we propose EasySOC (see Fig. 1). EasySOC takes as input an application where some of its constituent components have been implemented, and others are intended to be outsourced. In the figure, these two types of components are sketched with solid and dashed lines, respectively. Based on the Java interfaces describing the external components, a semi-automatic process is iteratively applied to associate an individual service with each one of these components. Each iteration involves three steps: (1) finding the list of candidate services, (2) selecting an individual service from the previous list, and (3) injecting a representative or proxy to the selected service into the application, to enable it to invoke the service at runtime. EasySOC provides developers with support tools that perform steps (1) and (3) automatically and semi-automatically, respectively, whereas step (2) is in charge of the software developer. For example, if a component for providing current foreign exchange rates is to be outsourced, ServiceObjects[2] and StrikeIron[3] services would be automatically discovered, one of these services selected by the developer, and a representative of the service integrated with the application. Overall, the discovery-selection-injection sequence is performed until all external components of the input application have been associated with a concrete service.

Typically, when manually looking for services that fulfill a certain functionality in a UDDI registry, a user first seeks a category related to that functionality, and then exhaustively analyzes the services that belong to it [9]. Essentially, the first step in Fig. 1 attempts to automatically reproduce this discovery process. EasySOC employs a Web Service search engine [10] that is based on a QBE approach and an automatic classifier [9]. Given a query or example, this search engine first deduces the most related category to the example functionality, and then looks for relevant services within it. Concretely, by analyzing the interface specification of a component C that is to be outsourced, EasySOC produces the example (sub-step 1.1 in Fig. 1) and sends it to the search engine (sub-step 1.2 in Fig. 1). As a result, though a large number of available services or categories may be present, a discoverer is allowed to promptly select a service from a wieldy list of candidates (step 2 in Fig. 1).

In order to *non-intrusively* integrate a selected Web Service with the consumer's application, EasySOC exploits the Dependency Injection (DI) [23] and Adapter design patterns. In DI terminology, when an application component C_1 needs the functionality of another component C_2, it is said that C_1 has a *dependency* to C_2. Then, the main goal of DI is to abstract away the code implementing dependencies (e.g. component instantiation and configuration) from the pure functional code implementing components, and to transparently inject the dependency code into components instead. By using DI, component code only depends on the interfaces describing components but not on the mechanisms by which application components communicate to each other. An interesting implication of DI to our work is that third-party services play the role of components to which internal components can depend upon, but without the need to explicitly provide functionality to actually invoke these services (i.e. Web Service APIs or frameworks). On the other hand, the implication of the Adapter design pattern is that application code neither depends on specific service contracts by adapting them to contracts expected by the internal components. In consequence, any internal component can take advantage of Web Services just like they were calling operations on another internal component, which makes service consumption more natural to the programmer, and frees the application logic from code that is tied to server-side service interfaces, which is semi-automatically injected and adapted by EasySOC instead (step 3 in Fig. 1).

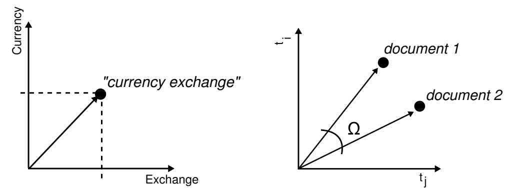
The remainder of this section will explain in detail the steps mentioned above. Particularly, the next subsection will focus on the first step of the outsourcing process, whereas Section 3.2 will concentrate on its second and third steps.

3.1. Discovering services

From an information retrieval viewpoint, the data within an information system includes two major categories: documents and queries. The key problems are how to state a query and how to identify documents that match that query [28]. The distinction between considering a query to be a document and considering it to be different from

[2]ServiceObjects `http://trial.serviceobjects.com/ce/CurrencyExchange.asmx?WSDL`
[3]StrikeIron `http://ws.strikeiron.com/ForeignExchangeRate?WSDL`

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013



(a) Representation (b) Comparison

Figure 2: Vector space model

a document affects the manner in which the retrieval process is modeled. If the query is considered to be a document, then retrieval is a matching process. The backbone of our service discovery approach is to use the same representation for both services and queries. Accordingly, the service discovery process is reduced to a matching problem.

Matching similar documents is a problem with a long history in information retrieval [28]. Methods based on linear algebra have shown to be suitable alternatives for correlating similar documents [12]. These techniques map documents onto a vector space (VS) [46]. Broadly, VS is an algebraic model for representing text documents in a multidimensional vector space, where each dimension corresponds to a separate term (usually single words). As a result, documents having similar contents are represented as vectors located near in the space. Moreover, a query is also represented as a vector. In consequence, searching related documents translates into searching nearest neighbors in a VS. For example, in Fig. 2 (a) we represent a document containing the terms "currency" and "exchange", whereas in Fig. 2 (b) the cosine of the angle Ω provides an estimation of how similar two vectors and therefore two documents are.

Essentially, our discovery technique deals with matching the interface of an external component to a concrete Web Service description. Then, the commented source code of the interface of a component being outsourced stands for a query, while vectors in the VS represent the descriptions accompanying available Web Services. Section 3.1.1 will explain in detail how vectors from client-side software are generated and Section 3.1.2 will describe how both spatial representations –i.e. client-side and server-side vectors– are matched.

3.1.1. Generating queries and mapping them onto the vector space

By automatically generating queries and narrowing the list of potential service candidates, EasySOC aims to ease the discovery task. The idea behind query generation is to extract relevant terms from the description (i.e. the Java interface) of a component being outsourced. In addition to the description of an external component, there are other sources of relevant terms that may be considered when building a query. Particularly, we assume that:

1. classes representing the parameters of an operation may contain relevant terms,
2. internal components interacting with the one being outsourced may contain relevant terms, this is, the source code context in which a service is invoked (e.g. a method) may also provide useful terms.

EasySOC expects good development practices from developers. In this way, we assume that, throughout their projects, developers use self-explanatory names for class properties, methods and arguments, comment them and avoid using meaningless names like "arg1", "arg2" or even the commonplace "foo", as as usually occurs [52]. Under these

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

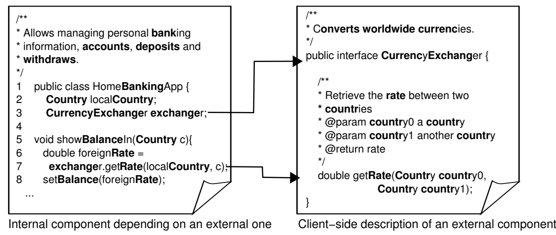


Figure 3: An example of relevant words within client-side commented source code

assumptions, method arguments of the interfaces describing external components may have meaningful terms. Moreover, the classes associated with these method arguments (e.g. the class Country in Fig. 3) may have proper names and documentation. In fact, this is expressed by the assumption number (1).

On the other hand, the assumption number (2) leads to extract relevant terms from those internal components that directly interact with the one being outsourced. Following good practices when building component-based software results in components with strongly-related and highly-cohesive operations [57]. Based on this fact, we assume that the logic of a well-designed application commonly belongs to a unique domain. For example, the right side of Fig. 3 depicts the documented Java interface describing an external component to get the currency exchange rate between two given countries and, on the left side, an internal component depending on it (line 3) and calling it (line 7). A Web Service for providing current foreign exchange rates might be useful for applications belonging to the business domain (the left side of Fig. 3 illustrates a home-banking application), while it rarely might be useful for an application in the math domain.

Java interfaces may contain terms that help to indicate their functionality. We define these terms as being relevant and other terms as non-relevant. In this way, all Java reserved words are non-relevant (e.g. public, void, interface, return). A Java interface comprises a name and a description of its provided operations (or method signatures in OO terminology). In addition, good development practices promote developers to comment source code. Javadoc[4] is a tool for automatically generating API documentation from comments in Java source code. With Javadoc developers place comments using a set of pre-established elements or tags. As a result, a Java interface specification consists of a structured textual description of its constituent parts (optional) and the signatures of its exposed operations (mandatory).

Java interfaces may contain terms that help to indicate their functionality. We define these terms as being relevant and other terms as non-relevant. In this way, all Java reserved words are non-relevant (e.g. public, void, interface, return). Extracting relevant terms is very important because they may contribute to build accurate queries, which in turn may help to increase the precision of the discovery mechanism as the next section will show. Consequently, we have designed a text mining process for extracting relevant terms from the client-side source code. This process comprises five activities. In a first activity, we pull out the name of a component and the name of its operations. To do this, we use the Java Reflection API[5]. Broadly, reflection provides

[4]Javadoc Tool Home `http://java.sun.com/j2se/javadoc`

[5]Java Reflection API. `http://java.sun.com/docs/books/tutorial/reflect/`

the ability to examine class meta-data [56]. In a second activity, we mine developers' comments from Javadoc elements. At this point, we have a collection of terms. Then, we look for combined words within this collection and split them, because commonly used notation conventions (e.g. JavaBean, Hungarian) suggest to combine two or more words (e.g. *getRate*, *get_rate* or *destCurrency*) for assigning names to operations and parameters. Finally, we employ Stop words and Stemming, two classic text mining techniques. A stop word is a word with a low level of "usefulness" within a given context or usage [28]. By removing symbols and stop words we attempt to "clean" queries. Finally, we utilize the Porter Stemming algorithm [41] for removing the commoner morphological and inflectional endings from words, reducing English words to their stems. As a result, the output of our text mining process is a set of stems extracted from the specification of the external component (e.g. the stems in bold in Fig. 3). Then, we use these stems for building a vector $\vec{q} = (e_0, \ldots, e_n)$, where each element e_i represents the weight of a distinct stem for the component being outsourced.

In [47] the authors compare different efforts that have been made on term weighting techniques. EasySOC uses TF-IDF because this combined heuristic has shown to be suitable for weighting terms present in Web Service descriptions [53]. TF determines that a term is important for a document if it occurs often in that document. On the other hand, terms which occur simultaneously in many documents are rated as less important because of their IDF value. Formally, for each term t_i of a document d, $tfidf_i = tf_i \bullet idf_i$, with:

$$tf_i = \frac{n_i}{\sum_{j=1}^{T_d} n_j} \qquad (1)$$

where the numerator (n_i) is the number of occurrences within d of the term being considered, and the denominator is the number of occurrences of all terms within d (T_d), and:

$$idf_i = \log \frac{|D|}{|\{d : t_i \varepsilon d\}|} \qquad (2)$$

where $|D|$ is the total number of documents in the corpus and $|\{d : t_i \varepsilon d\}|$ is the number of documents where the term t_i appears.

By employing this client-side text mining process on the descriptions of service operations and internal components, we augment the collection of terms that constitutes a query. In Section 4, we will evaluate how this approach impacts on the accuracy of the service discovery mechanism of EasySOC.

3.1.2. Matching similar queries and available Web Services

After generating a vector representation for a query, the next step is to match it against the vectors that stand for Web Services within the vector space to retrieve related services. In [9] we described how to map Web Service descriptions onto the VS. Broadly, we have developed a crawler that analyzes an UDDI registry, extracting the category and the WSDL document associated with each available service. Then, a WSDL document is preprocessed for extracting relevant terms and bridging syntactic differences of service descriptions. Specifically, the preprocessing stage for Web Services comprises extracting the names of the services, its operations and arguments along with any textual comment included in the WSDL document. Afterward, extracted terms are further refined by removing *stop-words*, employing Porter's *stemming* algorithm and bridging different WSDL message styles by mining relevant terms from data-type definitions. Finally, for each term we compute its $tfidf$-based weight and, in turn, the new vector is incorporated into the vector space.

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

Matching a query against the whole vector space can be very inefficient when the number of services is large [48]. Therefore, our search engine [10] uses a space reduction mechanism based on Rocchio's classification algorithm [22]. In [9] we have empirically shown that by using Rocchio with TF-IDF, this search engine achieves better results than using K-NN, Naïve Bayes and an ensemble machine learning approach [19] that combines Naïve Bayes and Support Vector Machine. This mechanism divides the vector space into sub-spaces, one for each category of services available in a UDDI registry. A sub-space is centered on an average vector, known as *centroid*, which stands for the documents that belong to that category. Afterward, a query is compared to the centroid associated with each category in order to determine the one that maximizes similarity. Once a category has been selected, the search engine compares the query *only* against the vectors that belong to this sub-space. This, besides being more efficient than matching a query against the whole vector space, reduces the number of dimensions of each individual sub-space [9] because services within an individual domain share the same sublanguage [32]. For the purposes of this paper we can informally define "sublanguage" as a form of natural language used in a sufficiently restricted setting [27]. Typically, a sublanguage uses only a part of the structures of a language. For instance, in the business domain words such as "economy", "competitive" and "currencies" occur often, while words such as "affine", "chebyshev" and "commutative" seldom appear. Formally, the centroid $\vec{c_i}$ for the documents that belong to category i is computed as:

$$\vec{c_i} = \alpha \frac{\sum_{\vec{d} \, \varepsilon \, C_i} \vec{d}}{|C_i|} - \beta \frac{\sum_{\vec{d} \, \varepsilon \, D - C_i} \vec{d}}{|D - C_i|}$$

with C_i being the sub-set of the documents from category i, and D the amount of documents of the entire data-set. First, both the normalized vectors of C_i, i.e. the positive examples for a class, as well as those of $D - C_i$, i.e. the negative examples for a class, are summed up. The centroid vector is then calculated as a weighted difference of the positive and the negative examples. The parameters α and β adjust the relative impact of positive and negative training examples. As suggested by [3], we use $\alpha = 16$ and $\beta = 4$.

There are some different similarity calculations for finding related vectors [28]. One measure that is widely used is the *cosine measure*, which has shown to be better than other similarity metrics in terms of retrieval effectiveness [26]. This measure is derived from the cosine of the angle between two vectors. This approach assumes that two documents with a small angle between their vector representations are related to each other. As the angle between the vectors shortens, its cosine approaches to 1, i.e. the vectors are closer, meaning that the similarity of whatever is represented by the vectors increases. Formally:

$$cosineSimilarity(Q, S) = \frac{Q \bullet S}{|Q||S|} = \frac{\sum_{i=1}^{T} t_{S,i} \times t_{Q,i}}{\sqrt{\sum_{i=1}^{T} t_{Q,i}^2 \sum_{i=1}^{T} t_{S,i}^2}}$$

We use this measure for matching a query Q against each service S, and then sort these results in decreasing order of cosine angles. The computational complexity of calculating cosine similarity between two vectors takes linear time and depends on the number of dimensions of the VS, i.e. the number of different terms T. In consequence, the

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

```
1: procedure DISCOVER(q⃗, N)                                  ▷ Returns a list of candidate Web Services
2:     Category[] category ← CLASSIFY(q⃗)
3:     for all v⃗_service ∈ category[0] do
4:         double similarity ← COSINESIMILARITY(q⃗, v⃗_service)
5:         INSERT(service, similarity, candidates)
6:     end for
7:     return SUBLIST(candidates, N)
8: end procedure
```

Algorithm 1: Main steps of the discovery process

space reduction mechanism reduces the time complexity of vector similarity calculations.

Algorithm 1 summarizes the main steps of the matching process for discovering relevant services. During the first step, the algorithm determines the nearest category of vector $\vec{q}$, which stands for a user's query (line 2). Afterward, the query is compared against each $\vec{v}_{service}$, i.e. the vector of a service that belongs to the category returned by the previous step (line 4). Found services are sorted according to cosine similarity (line 5), this is, vectors that minimize their angle between $\vec{q}$ are sorted first. Finally, the top N candidates are returned to the user (line 7).

For example, let us suppose there are 2 services belonging to a category named "book" and 2 services belonging to a category named "movie", whose corresponding vectors are:

$$\begin{aligned}
\vec{v}_0 &= (< book, 0.92 >, < searcher, 0.38 >) \\
\vec{v}_1 &= (< book, 0.86 >, < searcher, 0.35 >, < topic, 0.35 >) \\
\vec{v}_2 &= (< movie, 0.92 >, < topic, 0.38 >) \\
\vec{v}_3 &= (< movie, 0.86 >, < searcher, 0.35 >, < topic, 0.35 >)
\end{aligned}$$

Vectors $\vec{v}_0$ and $\vec{v}_1$ belong to category "book", whereas the other vectors belong to category "movie". Under our two-steps approach, the centroids for each category are:

$$\begin{aligned}
\vec{c}_{book} &= (< book, 0.93 >, < searcher, 0.34 >, < topic, 0.09 >) \\
\vec{c}_{movie} &= (< movie, 0.93 >, < topic, 0.34 >, < searcher, 0.09 >)
\end{aligned}$$

Now, let us suppose we want to find services for providing information about books covering a topic, by using "book topic" as input. Mapping the query onto this vector space generates a vector $\vec{q} =< book, 0.92 >, < topic, 0.38 >$. Then, EasySOC compares $\vec{q}$ against the aforementioned centroids (first step). The resulting similarities are:

$$\begin{aligned}
cosineSimilarity(\vec{q}, \vec{c}_{book}) &= 0.898 \\
cosineSimilarity(\vec{q}, \vec{c}_{movie}) &= 0.130
\end{aligned}$$

The centroid associated with "book" category maximizes the similarity, therefore EasySOC will compare the query only against $\vec{v}_0$ and $\vec{v}_1$ (second step). As a result, EasySOC performed 3 vector comparisons, instead of comparing $\vec{q}$ against the whole vector space. Moreover, as the reader can note the space has 4 dimensions: "book",

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

"searcher", "movie" and "topic". However, by reducing the search space, the number of terms was narrowed down to 3 during the second step ("book", "searcher" and "topic").

In the next section we will focus on describing in detail how discovered services are integrated with consumers' applications under EasySOC.

3.2. Incorporating a candidate

At step 3, after a developer selects a Web Service, EasySOC semi-automatically integrates the service with the application. To this end, EasySOC exploits the concept of Dependency Injection (DI). DI establishes a level of abstraction between application components via public interfaces, and achieves component decoupling by delegating the responsibility for component instantiation and binding to a DI container. In SOC terms, this represents the functionality for interpreting WSDL documents and performing calls to service providers.

Section 3.2.1 explains the concept of DI. Then, Section 3.2.2 describes how EasySOC builds on this notion to simplify Web Service consumption.

3.2.1. Dependency injection: Overview

Next, we will briefly illustrate DI through an example. Let us suppose we have a Java component for listing books of a particular topic (`BookLister`) that calls a remote Web Service-wrapped repository where book information is stored. The class implementing this component invokes the service operation that returns book information, and then iterates the results to filter and display this information:

```java
public class BookLister{
  private String endPoint = "http://example.edu:8080/BookRepository";
  private String ns = "http://example.edu";
  private String serviceName = "BookRepository";
  private String portName = "BookRepositoryPort";

  public BookLister(...){...}
  public void displayBooks(String topic){
    // Setup a call to the Web Service
    ServiceFactory sf = ServiceFactory.newInstance();
    Service service = sf.createService(new QName(ns, serviceName));
    Call call = (Call) service.createCall();
    call.setTargetEndpointAddress(endPoint);
    call.setPortTypeName(new QName(ns, portName));
    call.setOperationName(new QName(ns, "queryBooks"));
    call.setReturnType(new QName(NSConstants.NSURI_SCHEMA_XSD, "String[]"));
    // Contact the Web Service...
    Object wsResult = call.invoke(new Object[]{});
    List<Book> books = parseBooks((String[]) wsResult);
    Enumeration elems = books.elements();
    while (elems.hasMoreElements()){
      Book book = elems.nextElement();
      if (book.getTopic().equals(topic))
        System.out.println(book.getTitle() + ":" + book.getYear());
    }
  }
}
```

For clarity reasons, exception handling has been omitted. The `displayBooks` method contains two different types of instructions, namely, code to invoke the Web Service, and code to filter out the books that do not match the desired topic. Now, if we want to use a different mechanism for storing book information such as a database (i.e. no longer employ a Web Service to wrap the repository), `displayBooks` must be rewritten, some lines from `BookLister` discarded, and the whole component retested. Besides, depending on the way information is stored, a different set of configuration

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

parameters could be required (e.g. database location, drivers, etc.). In such a case, `BookLister` also have to be modified to include the necessary constructors/setters. Basically, the cause of this problem is that the implementation does not abstract away the API code for accessing the repository from the application logic, that is, the second group of instructions.

The DI-enabled listing component includes an interface (`BookSource`) by which `BookLister` accesses the repository. Classes implementing this interface represent a different form of accessing book information. In EasySOC terminology, such a class is called a *service adapter*. Additionally, `BookLister` exposes a `setSource(BookSource)` method so that a DI container can inject the particular retrieval component being used[6]. BookLister now contains code only for browsing and displaying book-related information, but the code which knows how and from where to obtain this information is placed on extra classes:

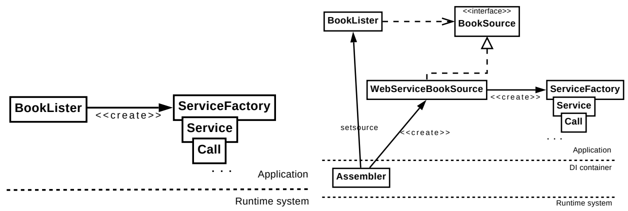
```java
/** The component into which another component is injected */
public class BookLister{
  BookSource source = null;

  public void setSource(BookSource source){ this.source = source; }
  public void displayBooks(String topic){
    List<Book> results = source.getBooks();
    // Filter and display results
  }
}
/** The interface of the dependency */
public interface BookSource{
  public List<Book> getBooks();
}
/** The component being injected */
public class WebServiceBookSource implements BookSource{
  private String endPoint = "http://example.edu:8080/BookRepository";
  private String ns = "http://example.edu";
  private String serviceName = "BookRepository";
  private String portName = "BookRepositoryPort";

  public void setEndPoint(String endPoint){ this.endPoint = endPoint; }
  public void setNS(String ns){ this.ns = ns; }
  public void setServiceName(String serviceName){ this.serviceName = serviceName; }
  public void setPortName(String portName){ this.portName = portName; }
  public List<Book> getBooks(){
    /**
     * 1) Setup a call to the Web Service
     * 2) Invoke its "queryBooks" operation
     * 3) transform the resulting array into a list object
     */
  }
}
```

Now, we must assemble the above components to build the whole application. Particularly, we have to indicate the DI container to use an instance of `WebServiceBookSource` for the `source` field of `BookLister`. This is supported in most containers by configuring a separate XML file, which specifies the DI-related configuration for every application component. From now on, we will use Spring [23] as the DI container. Then, the configuration file for the example is:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="myLister" class="BookLister">
```

[6] Many DI containers support two forms of injection: *setter injection* (components express dependencies via get/set accessors) and *constructor injection* (components express dependencies by means of constructor arguments).

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013



(a) Without using DI (b) Using DI

Figure 4: Class diagrams for the book listing application

```
      <property name="source"><ref local="mySource"/></property>
    </bean>
    <bean id="mySource" class="WebServiceBookSource"/>
      <property name="endPoint">http://example.edu:8080/BookRepository</property>
      <property name="ns">http://example.edu</property>
      <property name="serviceName">BookRepository</property>
      <property name="portName">BookRepositoryPort</property>
  </bean>
</beans>
```

Fig. 4 shows the class diagrams of the two versions of our book listing application. In the non-DI version (left), `BookLister` directly uses a Web Service API. Then, the application logic is mixed up with code for configuring and using Web Service protocols, thus reusability and extensibility suffer. Conversely, in the DI version (right), the code for contacting the service is encapsulated into a new component, and the corresponding configuration parameters are placed on a separate file, which is processed at runtime. As shown, using DI has reduced the number of dependencies to concrete classes within the application logic (i.e. `BookLister`) and produced a better design in terms of cohesion and extensibility.

Intuitively, the code implementing components is easier to reuse and to unit test, which in turn improves maintainability. For instance, `BookLister` and `WebServiceBookSource` can be separately modified, tested and reused. Empirically, it has been shown that software using DI tend to have lower coupling than software not employing DI [43], which has a direct impact on maintainability.

As shown, an interesting implication of DI in SOC is that the pure application logic can be isolated from the configuration details for invoking services (e.g. URLs, namespaces, port names, etc.). In fact, the Remoting module of Spring provides a number of built-in components that can be injected into applications to easily call services. Basically, this support makes Web Service invocation a transparent process. With this in mind, a developer thinks of a Web Service as any other regular component providing a clear interface to its operations. If a developer wants to call a Web Service S with interface I_S from within an internal component C, an external dependency between C and S is established through I_S, causing a proxy to S to be injected into C. This frees developers from explicitly using classes like `ServiceFactory`, `Service` and `Call` to invoke Web Services.

This development practice, which can be seen as a contract-first approach to Web Service consumption, effectively leverages the benefits of DI for building service-oriented software. However, it leads to a form of coupling through which the application is tied to the contracts (i.e. the I_S interface) of the specific services it relies on. In this way, changing the provider for a service requires to adapt the client application to follow the new service contract. At the implementation level, this means to rewrite the portions of the application code that use the interface of the original service. A

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

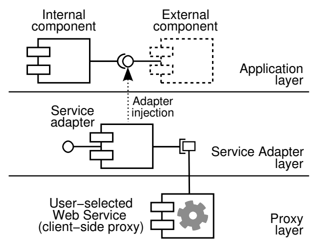


Figure 5: Service adapters in EasySOC

different interface implies different operation names, and input and return data-types (e.g. a complex data-type array instead of `String[]` for our book service), which must be adapted manually. All in all, the DI pattern is useful for building loosely coupled components. However, when using a contract-first approach to service consumption, DI may not be enough to ensure modifiability in the resulting software.

3.2.2. Taking DI a step further

To overcome this problem, EasySOC refines the idea of Web Service injection by introducing an intermediate layer that allows applications to non-invasively use services. Roughly, instead of directly injecting a raw Web Service proxy into the application, a *service adapter* is injected (see Fig. 5). A service adapter is a specialized Web Service proxy, inspired by the Adapter design pattern [16], which is in charge of adapting the interface of the underlying service according to the interface (specified by the developer at design time) of the associated external component. Service adapters comprise the logic to transform the method signatures of the external component (i.e. the client-side interface used by EasySOC as a query to perform service discovery) to the actual interface of the Web Service selected by the developer. For instance, if a service operation returns multiple integers as a comma-tokenized string, but the application requires an integer array, the adapter would be responsible for performing the conversion.

In opposition to the contract-first approach to outsourcing, in which the application code is made compatible with the interfaces of the services it uses, service adapters accommodate the interfaces of the outsourced services to the interfaces supplied by the developer. This approach is called *code-first*. Then, changing a service does not affect the code of the application, because it only requires to write a different service adapter for the new service. Besides reducing the coupling between internal components of an application and services, this approach allows developers to design, implement and test the application components, and then focus on the "servification" of the application. Furthermore, this separation may bring additional benefits beyond software quality and contribute to improve the development process itself, as these two groups of tasks can be performed independently by different development teams.

To better illustrate these ideas, and to understand the responsibilities of the developer in the tasks of incorporating a candidate service for an external component, let us come back to the DI version of the book listing application discussed above. Let us suppose our application is now composed of an internal component (`BookLister`) and an external component, whose contract is specified by the `BookSource` interface and for which we want to outsource an implementation. Based on the example (`BookSource`),

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

EasySOC[7] automatically retrieves the WSDL locations of the candidate services. After the developer has chosen a service from this list, EasySOC generates a proxy to the service, the corresponding service adapter, and the DI configuration to inject these two components into the application.

The proxy to the selected Web Service is created based on its WSDL description, and holds the necessary logic to talk to the service. The interface of the proxy is exactly the same as the service contract established by the particular provider, which, under a code-first approach to service outsourcing, will not usually be truly compliant to the service contract expected by the application (in our case `BookSource`). Currently, proxy generation is based on the Web Tools Platform (WTP) project[8]. Then, the service adapter is partially generated by EasySOC. It is implemented as a class skeleton that bridges the interface of the client-side proxy to the service contract expected by `BookLister`. Since the adapter is injected into `BookLister`, it realizes `BookSource`, that is, the interface of the component being outsourced. The actual code to forward any call to methods from this skeleton class to the proxy must be implemented by the developer. For instance, let us assume that the interface of the generated proxy is:

```
public interface BookSource_Proxy{
  public BookInfo[] getStoredBooks();
}
```

, where `getStoredBooks` is an operation derived from the WSDL description of the Web Service. Then, the adapter must map individual calls to `getBooks` (application-level contract) to calls to `getStoredBooks` (server-side contract) on the proxy, thus the final service adapter code would be:

```
public class BookSource_Adapter implements BookSource{
  private BookSource_Proxy proxy = null;

  public void setProxy(BookSource_Proxy proxy){ this.proxy = proxy; }
  public BookSource_Proxy getProxy(){ return proxy; }
  public List<Book> getBooks(){
    Vector<Book> expected = new Vector<Book>();
    BookInfo[] adaptee = getProxy().getStoredBooks();
    for (int i=0; i<adaptee.length; i++)
      expected.addElement(new Book(adaptee[i].getTitle(), adaptee[i].getYear()));
    return expected;
  }
}
```

The service adapter only implements the translation of the invoked operation name and its return data-type. However, the mapping task may also involve converting the input arguments of one or more adapter operations to the parameters of proxy operations. Besides, adapters are useful for including extra operation arguments that otherwise would be in the application code (e.g. username/password, licensing information, etc.). In addition, using adapters isolates the application logic from the code for handling service-related exceptions.

Finally, EasySOC creates the DI-related configuration to wire the service proxy, the adapter and the internal component(s) using the Web Service together by automatically appending the extra component definitions to the XML configuration of the application (see Fig. 6 (a)). The configuration tells the DI container to inject an instance of the generated adapter into the corresponding internal component (`BookLister`), and also a

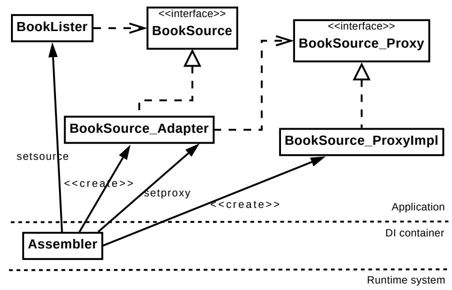
[7]The development of a plug-in for the Eclipse SDK providing graphical tools to simplify as much as possible the whole outsourcing process is underway

[8]The Web Tools Platform `http://www.eclipse.org/webtools`

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

```
...
<beans>
  <bean id="myLister" class="BookLister">
    <property name="source">
      <ref local="source_Adapter"/>
    </property>
  </bean>
  <!-- Service adapter -->
  <bean id="source_Adapter" class="BookSource_Adapter">
    <property name="proxy">
      <ref local="source_Proxy"/>
    </property>
  </bean>
  <!-- Client-side proxy to the WebService -->
  <bean id="source_Proxy" class="BookSource_ProxyImpl"/>
</beans>
```

(a) DI-related configuration



(b) Class diagram

Figure 6: The EasySOC book listing application

proxy to the service (`BookSource_ProxyImpl`) into the service adapter. The class diagram for the entire application is shown in Fig. 6 (b). In general terms, a service proxy can be associated with only one adapter, but the same adapter may be indirectly used by more than one internal component, this is, when many implemented components depend on the same external component.

4. Evaluation

This section describes the experimental evaluation of EasySOC. The next subsection details the evaluation of its discovery mechanism. Then, subsection 4.2 will concentrate on evaluating its programming model for service consumption.

4.1. Evaluation of the discovery mechanism

In [9], we specifically discussed the accuracy of the classification mechanism of EasySOC through different tests. Therefore, we focus here on analyzing the effectiveness of the query generation phase. Concretely, we analyzed the implications of generating queries using terms extracted from different parts of 30 client applications by using the *R*-precision, Recall and Precision-at-*n* measures [28]. In addition, we evaluated the effort demanded in discovering services with and without the assistance of EasySOC.

As we mentioned in Section 3.1.1, EasySOC extracts relevant terms from the description of an external component that is to be outsourced. Basically, there may be four different sources of terms associated with a component description: (1) its functional interface, (2) its documentation, (3) the classes of its operation arguments, and (4) the classes of those components that directly interact with it. We named the first source “Interface”. When using this source, we just considered the name of a component along with the names of its operations. We did not take into account natural language descriptions (e.g. Javadoc comments) in the queries. In fact, we focused on measuring the performance of the discovery mechanism with very short descriptive queries. Conversely, when incorporating the second source, we extracted terms from the Javadoc comments of the external component description as well. We named the combination of sources 1 and 2 “Documentation”. In addition, we used the third source to consider the name and the Javadoc comments found in the classes associated with the operation arguments, i.e. if an argument type is non-primitive then we mined terms from its class. We called the combination of sources 1, 2 and 3 “Arguments”. Finally, by adding

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

Table 1: Number of different stems extracted per query

1. (4,7,20,20)	6. (4,7,7,19)	11. (4,6,31,13)	16. (3,7,18,34)	21. (5,10,11,21)	26. (3,6,10,18)
2. (3,5,5,16)	7. (4,8,12,29)	12. (1,5,5,10)	17. (5,8,18,22)	22. (3,11,21,21)	27. (5,8,37,31)
3. (4,9,12,17)	8. (3,6,12,11)	13. (4,7,23,12)	18. (5,8,12,28)	23. (3,8,10,18)	28. (4,11,17,19)
4. (3,6,13,20)	9. (4,6,31,13)	14. (4,9,9,22)	19. (3,8,15,32)	24. (4,8,10,18)	29. (5,11,13,29)
5. (5,15,15,20)	10. (4,8,16,23)	15. (2,7,9,18)	20. (4,8,17,21)	25. (4,7,11,24)	30. (6,15,32,31)

source 4 to sources 1 and 2, we collected terms from the name and Javadoc comments associated with the classes of those internal components that directly depend on the one being outsourced. We called the combination of sources 1, 2 and 4 “Dependants”.

To perform the tests and feed our discovery system, we used a publicly available collection of categorized Web Services [19]. The data-set comprises 391 WSDL documents divided in 11 categories. We preprocessed each WSDL document according to [9], thus resulting in a vector of relevant stems per Web Service. As shown in [2], in general several naming tendencies take place in WSDL documents. For example, the authors found that a message part standing for a user's name, is called in many syntactically different ways, e.g. “name”, “lname”, “userName” or “first_name” [2]. When building the vector space, our search engine deals with these tendencies. For example, initially there were 7548 unique words within the WSDL documents of the “financial” category, however there were 2954 after preprocessing the service descriptions [9].

Moreover, we built 30 queries to use them as the evaluation-set. Each query was written in Java and consists of an interface describing the functional capabilities of an external component and an internal component that used it. We commented both the header and the operations of the interface. Besides, for those operations that used non-primitive data-types as arguments, we also commented their corresponding classes. Each query is associated with a four-tuple, representing its size in terms of the number of stems that resulted from processing its related sources (see Table 1) . For instance, the 7^{th} query, which results after preprocessing the source code showed in Fig. 3, consists of 4 stems: “countri”, “currenc”, “exchang”, “rate”, when mining terms only from the interface of the external component expected at the client-side (source 1). The query comprised 8 different stems when incorporating the stems “convert”, “retriev”, “tool” and “worldwid” from both sources 1 and 2. When adding “divis”, “entiti”, “geograph” and “polit” from the descriptions of its operation arguments (sources 1, 2 and 3) the query comprised 12 stems. When combining sources 1, 2 and 4 the query consisted of 29 stems, incorporating the stems “bank”, “transfer”, “destini”, “origin”, “allow”, “balanc”, “class”, “client”, “current”, “histori”, “method”, “monei”, “mount”, “page”, “repres”, “transact” and “sale”. Therefore, the four-tuple for the aforementioned query is $(4, 8, 12, 29)$.

There are some different methods for evaluating the performance of a retrieval system. We decided to measure the performance of our discovery mechanism in terms of the proportion of relevant services in the retrieved list and their positions relative to non-relevant ones. In this sense, we employed R-precision, Recall and Precision-at-n measures. An important characteristic regarding the present evaluation is the definition of “hit”, i.e. when a returned WSDL document is actually relevant to the user. During the tests, a software developer judged the retrieved documents in response to each query: if he determined that the operations of a retrieved WSDL document fulfilled the

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

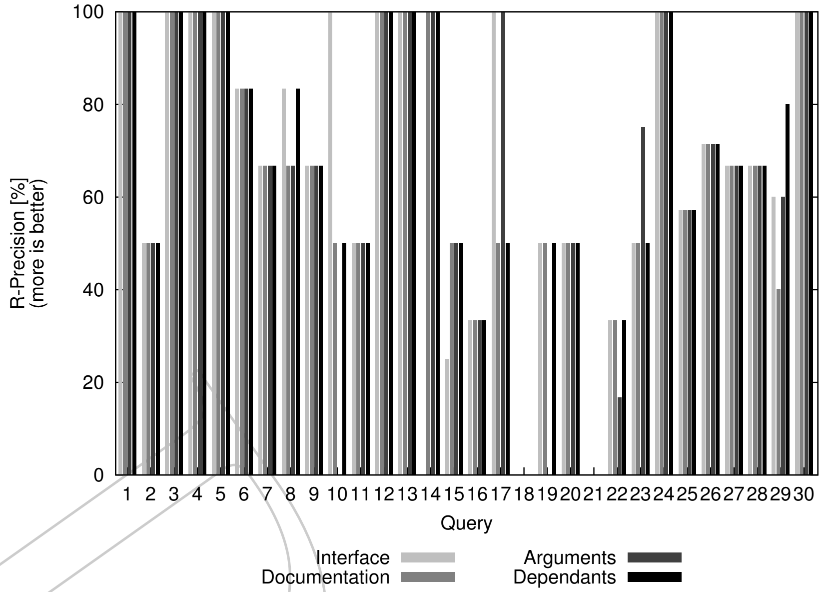


Figure 7: *R*-Precision of the experiments

expectations previously specified in the Java code, then a hit was produced. For example, if he expected an operation for converting from Euros to Dollars, then a retrieved operation for converting from Francs to Euros was non-relevant, even though these operations belonged to the same category or they were strongly related. In this particular case, only operations for converting from Euros to Dollars were relevant. Note that this definition of hit makes the validation of our discovery mechanism more strict than previous efforts.

4.1.1. R-*precision*

One of the most used measures for assessing retrieval performance is *R*-precision. Basically, given a query with *R* relevant documents, this measure computes the precision at the R^{th} position in the ranking ($RetRel_R$). For example, if there are 10 documents relevant to the query within the data-set and they are retrieved before the 11^{th} document, we have a *R*-precision of 100%, but if 5 of them are retrieved after the top 10 we have 50%. Formally, $R\,precision = \frac{RetRel_R}{R}$. We obtained the *R*-precision for the above 30 queries by individually using each one its four combinations of sources of terms (a total of 120 experiments). Fig. 7 depicts the achieved *R*-Precision of each experiment. The average *R*-precision of the Interface, Documentation, Arguments and Dependants combinations were 65.45%, 65.06%, 64.34% and 66.95%, respectively. These percentages were computed by averaging each set of results over the 30 queries.

It is worth noting that for any query there are, at most, 8 relevant services within the data-set. Besides, there are 10 queries that have associated only one relevant service. This particularity of the data-set severely harms the precision of our discovery mechanism when the first retrieved service is not relevant. For instance, the query number 18 had only one relevant service within the data-set, which was ranked sixth in the four candidate lists. Hence, *R*-precision of this query was $0 = \frac{0}{1}$. In spite of the described situation, the overall results show that, when using the Dependants combi-

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

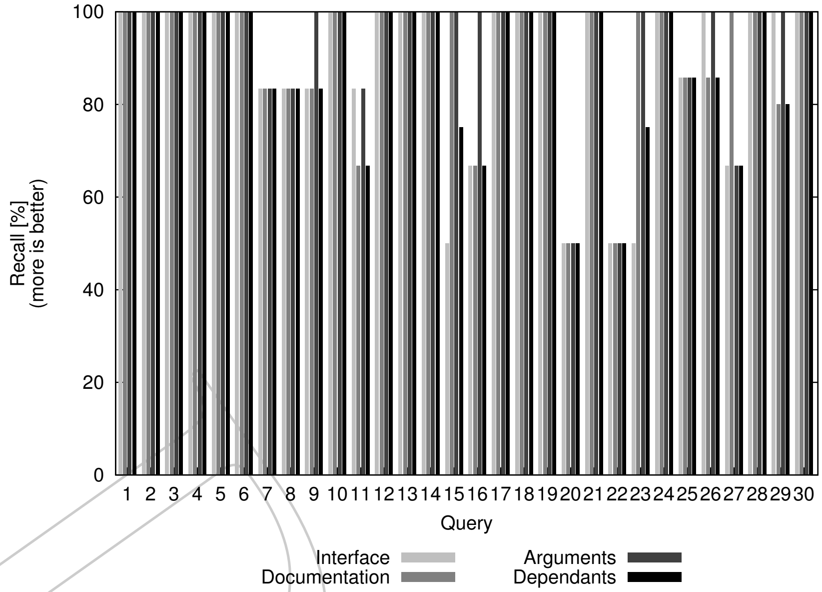


Figure 8: Recall of the experiments

nation, EasySOC included at the average 66.95% of the relevant services at the top of the list. This means that EasySOC included nearly 67% of the relevant services before non-relevant services.

4.1.2. *Recall*

Recall is a measure of how well a search engine performs in finding relevant documents [28]. Recall is 100% when every relevant document of a data-set is retrieved. Formally, $Recall = \frac{RetRel}{R}$ where $RetRel$ is the total number of relevant services included in the list of candidates. By blindly returning all documents in the collection for every query we could achieve the highest possible recall, but looking for relevant services in the entire collection is clearly a slow task. In addition, we want to achieve good Recall in a window of *only* 10 retrieved services. We have chosen this window size because we want to balance between the number of candidates and the number of relevant candidates retrieved and we believe that a developer can certainly examine 10 Web Service descriptions without much effort. Therefore, we measured the Recall for each query by setting $RetRel = RetRel_{10}$. Again, we computed the Recall for the 120 experiments and then we averaged the results. The average Recalls of the Interface, Documentation, Arguments and Dependants were 88.41%, 91.16%, 93.41% and 88.38%, respectively. Fig. 8 depicts the achieved Recall of each experiment. Graphically, all Recall values (y-axis) are greater than 0, i.e. EasySOC included, at least, one relevant service for every query in the top 10 retrieved services.

4.1.3. *Precision-at-*n

Precision-at-n measure computes precision at different cut-off points [28]. For example, if the top 10 documents are all relevant to a query and the next 10 are all non-relevant, we have a precision of 100% at a cut-off of 10 documents but a precision of 50% at a cut-off of 20 documents. Formally, $Precision\,at\,n = \frac{RetRel_n}{n}$ where

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

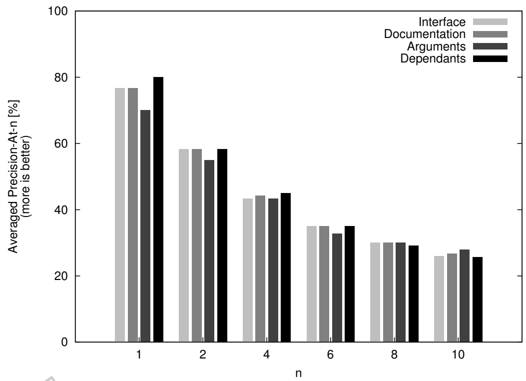


Figure 9: Average Precision-at-n

$RetRel_n$ is the total number of relevant services retrieved in the top n. We evaluated Precision-at-n for each query when using the aforementioned combinations of sources and averaged the results. We measured by using $n = 1, 2, 4, 6, 8, 10$. Fig. 9 shows the average Precision-at-n of the experiments. Once again, the number of relevant services per query within this particular data–set harms the precision of our discovery approach as n and the amount of retrieved services increases. Nevertheless, the results show that 80% of the services at the top of the candidate list were relevant when employing Dependants. Furthermore, using Arguments, Precision-at-*1* was 70%. Both Interface and Documentation combinations resulted in a Precision-at-*1* of 76.67%.

4.1.4. Discussion

During a typical discovery process (i.e. without EasySOC) a discoverer usually tries to deduce the category of the desired service, so as to reduce the search space. Afterward, the discoverer examines the services that belong to the deduced category. Although each category has its own service population, the most populated category in the data-set used in the evaluation has 65 services, and there is an average of 40 services per category. Therefore, we estimate that discovering services with this data-set has a cost of 65 and 40 WSDL documents per query on the worst and average cases, respectively. Here, the cost associated with an individual WSDL document may be the time spent by the user in examining it to determine whether it is relevant.

The achieved Recall results have shown that by using EasySOC a discoverer usually selects a proper service from a set of only 10 WSDL documents. In fact, this set is an ordered list where services having a higher confidence of being relevant to the query are located at the top, as shown by the achieved R-precision and Precision-at-*1* results. As a consequence, the user sequentially examines, at worst, 10 WSDL documents before finding one relevant service. We measured the average position of the first relevant services within the retrieved candidate services, which resulted in 1.73, 1.7, 1.8 and 1.6 using Interface, Documentation, Arguments and Dependants combinations, respectively. Therefore, a discoverer examines only 2 WSDL documents on the average case, and 10 WSDL documents on the worst case, for our data-set. In other words, EasySOC has reduced the cost of the discovery process over the data-set by 95% (average case) and 85% (worst case) with respect to doing the same task without any assistance. Clearly, although these results can not be generalized to other data-sets, they are promissory.

4.2. Case study: A personal agenda software

In the next paragraphs we detail a comparison between the implementation of a service-oriented application based on both the contract-first approach to service engagement (i.e. coding the application logic comes *after* knowing the contract of the external services to be consumed) and EasySOC. Basically, we separately used these two alternatives to develop a simple, service-based personal agenda software using some of the Web Services of the aforementioned data-set. Unlike the previous section, the purpose of the evaluation described in this section is not to assess the effectiveness of EasySOC when discovering Web Services, but quantifying the source code quality resulting from employing either contract-first or EasySOC for actually consuming the discovered services.

After implementing the logic, incorporating the Web Services, and testing each version of the application, we randomly picked one service already incorporated into the applications and we changed its provider. Then, we took metrics on the resulting source codes in an attempt to have an assessment of the benefits of EasySOC for software maintenance with respect to the contract-first approach. For simplicity reasons, the analysis ignored the code implementing the GUI of the personal agenda software. Data collection was performed by using the Structure Analysis Tool for Java (STAN) [9].

The main responsibilities of the personal agenda software is to manage a user's contact list and to notify these contacts of events related to planned meetings. The contact list is a collection of records, where each record keeps information about an individual, such as name, location (city, state, country, zip code, etc.), email address, and so on. Below is the list of tasks that are carried out by the application upon the creation of a new meeting. We assume the user provides the date, time and participants of the meeting, as well as the location where the meeting will take place. Also, we simplify the problem of coordinating a realistic meeting by assuming that the participants being notified always agree with the arrangement provided by the user of the personal agenda software. In summary, the notification process roughly involves:

- **Getting a weather forecast** for the meeting place at the desired date and time.
- **Obtaining the routes** (driving directions) that each contact participating in the meeting could employ to travel from their own location to the meeting place.
- For each participant of the meeting:
  - Building an email message with an appropriate subject, and a body including the weather report and the corresponding route information.
  - **Spell checking** the text of the email.
  - **Sending the email**.

The text in bold represent the functionalities that were outsourced to Web Services during the implementation of the different variants of the application. As the contract-first approach does not assist developers in finding services, each Web Service was discovered using our search engine along with four of the queries shown in Table 1. Specifically, we queried the search engine for a weather forecaster service (query #29), a route finder service (query #10), a spellchecker service (query #24), and an email sender service (query #22). We followed the text mining process described in Section 3.1.1 to

[9]Structure Analysis for Java `http://www.stan4j.com`

build these queries from the client-side interfaces of the EasySOC implementation of the personal agenda software. Once the Web Services were discovered, we used their corresponding WSDL documents as the outsourced services for the contract-first application.

The following list summarizes the metrics that were taken on the resulting application code:

- *SLOC (Source Lines Of Code)* counts the total non-commented and non-blank lines across the entire application code[10], including the code implementing the pure application logic, plus the code for interacting with the various Web Services. The smaller the SLOC value, the less the amount of source code that is necessary to maintain once an application has been implemented. Since the present evaluation specifically aims at assessing the technical quality of the source code of the applications, class documentation was left out of the scope of the analysis.

- *Ce (Efferent Coupling)*, indicates how much the classes and interfaces within a package depend upon classes and interfaces from other packages [33]. In other words, this metric includes all the types within the source code of the target package referring to the 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, 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 is the influence of the adapter layer of EasySOC on this kind of dependency.

- *CBO (Coupling Between Objects)* is the amount of classes to which an individual class is coupled [6]. 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 [21], CBO can be used as a complementary indicator of how maintainable a software is.

- *RFC (Response for Class)* 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 [6]. 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 [21], it is highly desirable to achieve low RFC values for application classes.

Table 2 shows the resulting metrics for the four implementations of the personal agenda software: contract-first, EasySOC, and two additional variants in which another provider for the weather forecaster service was chosen from the Web Service data-set. For convenience, we labeled each implementation with an identifier (*id* column), which

[10] As defined in the COCOMO cost estimation model

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

Table 2: Personal agenda software: source code metrics

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

will be used through the rest of the paragraphs of this section. To perform a fair comparison, the following tasks were carried out on the final implementation code:

- The source code was transformed to a common formatting standard, so that sentence layout was uniform across the different implementations of the application. This, together with the fact that only one person was involved in the implementation of the applications, minimizes the impact of different coding conventions that may bias the values of the metrics that depend on the number of lines of source code.

- Java import statements within compilation units were optimized by using the source code optimizing tool of the Eclipse SDK. Basically, this tool automatically resolves import statements, thus leaving in the application code only those classes which are actually referenced by the application.

- In every implementation of the application the client-side proxies to the Web Services were exactly the same (generated through Eclipse WTP). Consequently, their associated source code was not considered for computing the aforementioned metrics.

4.2.1. Discussion

From Table 2, 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 relying on EasySOC (E_x), did not incur in extra efferent couplings with respect to the variants implemented according to the contract-first approach (C_x). Furthermore, if we do not consider the corresponding service adapters, Ce for the EasySOC variants drops down to zero, because EasySOC effectively pushes the code that depends on service contracts out of the application logic.

Fig. 10 shows the resulting SLOC. As the reader can see, changing the provider for the weather forecaster 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 contract-first approach, the incorporation of the new service provider caused the modification of 17 lines from C_1 (more than 7% of its code).

Note that the variants coded under EasySOC had an SLOC greater than that of the variants based on the contract-first approach. However, this difference was caused by the code implementing service adapters. In fact, the non-adapter code was smaller, cleaner and more compact because, unlike its contract-first counterpart, it did not include statements related to importing and instantiating proxy classes and handling Web Service-specific exceptions. Additionally, there are positive aspects concerning service

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

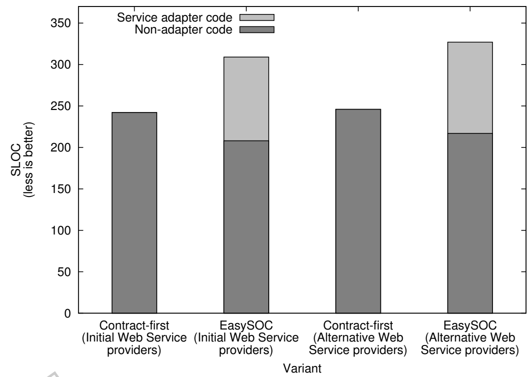
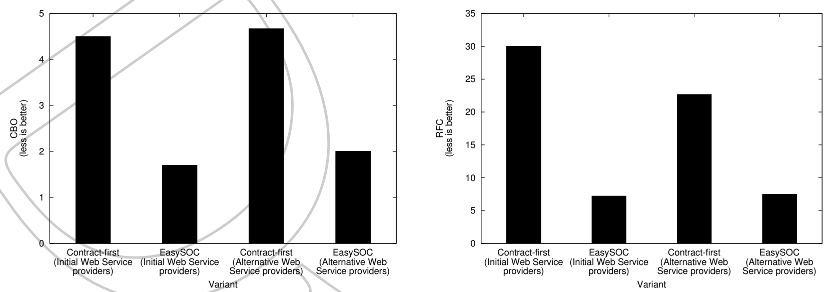


Figure 10: Source Lines of Code (SLOC) of the different applications



(a) CBO (b) RFC

Figure 11: Coupling Between Objects (CBO) and Response for Class (RFC) of the different applications

adapters and SLOC. On one hand, a large percentage of the service adapter code was generated automatically, which means programming effort was not required. On the other hand, changing the provider for the weather forecaster 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 contract only in the newly generated adapter class. Conversely, replacing the forecaster 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 browse and modify much more code. In addition, this practice might have introduced more bugs into the already tested application.

As mentioned earlier, CBO and RFC metrics were also computed (Fig. 11). Particularly, high CBO is extremely 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 (change-prone) service contracts, should be kept to a minimum. Similarly, low RFC implies better testability and debuggability. In concordance with Ce, which resulted in greater values for the modified variants of the application, CBO for both EasySOC and contract-first exhibited increased values when changing the provider for the forecaster service. On the other hand, RFC presented a less uniform behavior.

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

As reported by the Ce metric, EasySOC did not reduce the amount of efferent couplings from the package implementing the application logic. Naturally, the reason of this fact is that the service contracts adhered by E_x are exactly the same as C_x. However, the EasySOC applications reduced the CBO with respect to the contract-first implementations, because the access to the various services utilized by the application, and therefore their associated data-types, is performed within several cohesive compilation units (i.e. adapters) rather than within few, more general classes. This approach improves reusability and testability, since application logic classes do not directly depend on services.

As depicted in Fig. 11 (b), this separation also helped in achieving better average RFC. Moreover, although the plain sum of the RFC values of the E_x were greater compared to C_x, the total RFC of the classes implementing application logic (i.e. without taking into account adapter classes) were both smaller. This suggests that the pure application logic of E_1 and E_2 is easier to understand than C_1 and C_2. In large projects, we reasonably may expect that much of the source code of EasySOC applications will be part of the application logic instead of service adapters. Therefore, preserving the understandability of this kind of code is crucial.

5. Conclusions

We have presented EasySOC, a new approach to simplify the development of service-oriented applications. Among the strengths of EasySOC is its novel mechanism for accurately and efficiently discovering existing Web Services based on machine learning techniques, and a convenient programming model based upon the concept of Dependency Injection that allows developers to non-invasively consume external services. Concretely, the aim of EasySOC is to exploit the information present in client-side source code to ease the task of discovering services, and at the same time let programmers to separate the application logic from service-related concerns in order to increase the maintainability of the resulting software.

We have shown the benefits of EasySOC for building Web Service-based applications through a number of experiments. Specifically, we evaluated the retrieval effectiveness of its discovery mechanism by comparing four different heuristics for automatic query generation from source code on a data-set of 391 Web Services. Moreover, we assessed the advantages of EasySOC with regard to software maintainability through several applications that consumed services from this data-set and source code metrics. Our preliminary findings are very encouraging. With respect to service discovery, all heuristics achieved a recall in the range of 88-94%, which means that a high percentage of relevant services are retrieved. Furthermore, for some heuristics, we obtained a precision-at-*1* (i.e. the first retrieved service is always relevant) of around 75-80% at the average. We also showed that using different portions of the client-side code for generating queries can help in improving the performance of our discovery mechanism. With respect to service consumption, we found that, at least for the analyzed applications, using EasySOC led to software whose functionality was fully isolated from common service-related concerns, such as interfaces, data-type conventions, protocols, etc. For the discussed applications, as reported by the well-established CBO and RFC metrics, the EasySOC implementations also achieved better coupling and cohesiveness than the software built under the contract-first approach.

However, despite the above results, we will conduct more experiments to further validate EasySOC. We will evaluate the performance of our discovery mechanism with other data-sets. As a starting point, we will use a recently published collection of

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

real Web Services[11]. Second, we are also planning to use EasySOC for developing larger applications. Note that this might enable the use of metrics specially designed to quantify software quality and maintainability in large projects like the Maintainability Index [8] or the metrics suite proposed in [30]. In addition, we could employ different development teams so as to consider human factors in the assessment as well.

EasySOC is a technology-agnostic approach to Web Service discovery and consumption. In fact, many of the technological details discussed throughout this paper should be thought as being part of just one materialization of EasySOC out of many alternatives. On one hand, the first step of our outsourcing process (i.e. service lookup) can be extended to support different service description language (e.g. WSDL, CORBA-like IDLs, etc.), many registry infrastructures (e.g. UDDI, CORBA), different intermediate representations when extracting terms from source code (e.g. reflection, syntax tree, etc.) and various programming languages. Similarly, the third step of this process (service engagement) can be implemented for any programming language that has support for DI and Web Service proxying. Currently, several DI and Web Service frameworks for a variety of languages already exist (C++, Python, Ruby, etc.).

This work will be extended in several directions. With respect to our search engine, we will experiment with other weighting schemes. Specifically, term distributions [31] and TF-ICF [44] have shown promissory results, but they have not been used in the context of Web Services yet, at least, to the best of our knowledge. Another line of research involves the provision of some assistance to developers for programming service adapters. As mentioned before, we could use a technique similar to [37] to partially automate the task of bridging the signatures of the methods declared by an adapter and the operations of its associated Web Service. Another interesting work is concerned with taking into account some of the runtime aspects of Web Services in the outsourcing process. For instance, unpredictable runtime conditions (e.g. network or software failures) can degrade the performance of Web Services or even cause them to become unavailable, which in turn affect the execution of those EasySOC applications that rely on failing services. To overcome this problem, we will enhance service adapters to support "hot-swapping" of services alternatives. Specifically, rather than representing only one Web Service, individual adapters will maintain a list of candidate services. Therefore, at runtime an adapter will be able to choose between different service implementations according to different criteria (availability, performance, throughput, etc). Of course, this solution increases the cost of writing adapters, since more code to accommodate adapter method signatures and Web Service operations have to be provided. In this sense, assisting developers in this task will be crucial.

Acknowledgments

We deeply thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

[11]The QWS Dataset `http://www.uoguelph.ca/~qmahmoud/qws/index.html`

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

References

[1] M. B. Blake, D. R. Kahan, M. F. Nowlan, Context-aware agents for user-oriented Web Services discovery and execution, Distributed and Parallel Databases 21 (1) (2007) 39–58.

[2] M. B. Blake, M. F. Nowlan, Taming web services from the wild, IEEE Internet Computing 12 (5) (2008) 62–69.

[3] C. Buckley, G. Salton, J. Allan, The effect of adding relevance information in a relevance feedback environment, in: 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '94), Dublin, Ireland, Springer-Verlag, New York, NY, USA, 1994.

[4] M. Burstein, C. Bussler, M. Zaremba, T. Finin, M. N. Huhns, M. Paolucci, A. P. Sheth, S. Williams, A semantic Web Services architecture, IEEE Internet Computing 9 (5) (2005) 72–81.

[5] L. Cavallaro, E. Di Nitto, An approach to adapt service requests to actual service interfaces, in: 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08), Leipzig, Germany, ACM Press, New York, NY, USA, 2008.

[6] S. R. Chidamber, C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[7] M. A. Cibrán, B. Verheecke, W. Vanderperren, D. Suvée, V. Jonckers, Aspect-oriented programming for dynamic Web Service selection, integration and management, World Wide Web 10 (3) (2007) 211–242.

[8] D. Coleman, D. Ash, B. Lowther, P. Oman, Using metrics to evaluate software system maintainability, Computer 27 (8) (1994) 44–49.

[9] M. Crasso, A. Zunino, M. Campo, AWSC: An approach to Web Service classification based on machine learning techniques, Inteligencia Artificial, Revista Iberoamericana de IA 12 (37) (2008) 25–36.

[10] M. Crasso, A. Zunino, M. Campo, Query by example for Web Services, in: 2008 ACM Symposium on Applied Computing (SAC '08), Fortaleza, Ceara, Brazil, ACM Press, New York, NY, USA, 2008.

[11] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana, The next step in Web Services, Communications of the ACM 46 (10) (2003) 29–34.

[12] S. Deerwester, S. T. Dumais, G. W. Furnas, T. Landauer, R. Harshman, Indexing by latent semantic analysis, Journal of the American Society for Information Science 41 (6) (1990) 391–407.

[13] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, J. Zhang, Similarity search for Web Services, in: 30th International Conference on Very Large Data Bases, Toronto, Canada, Morgan Kaufmann, 2004.

[14] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology, and Design, Prentice Hall, Upper Saddle River, NJ, USA, 2005.

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

[15] D. Fensel, H. Lausen, J. de Bruijn, M. Stollberg, D. Roman, A. Polleres, Enabling Semantic Web Services: The Web Service Modelling Ontology, Springer-Verlag, Secaucus, NJ, USA, 2006.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, USA, 1995.

[17] J. D. Garofalakis, Y. Panagis, E. Sakkopoulos, A. K. Tsakalidis, Contemporary Web Service discovery mechanisms, Journal of Web Engineering 5 (3) (2006) 265–290.

[18] A. Gomez-Perez, O. Corcho-Garcia, M. Fernandez-Lopez, Ontological Engineering, Springer-Verlag, Secaucus, NJ, USA, 2003.

[19] A. Heß, E. Johnston, N. Kushmerick, Assam: A tool for semi-automatically annotating semantic Web Services, in: 3rd International Semantic Web Conference (ISWC2004), Hiroshima, Japan, vol. 3298 of Lecture Notes in Computer Science, Springer, 2004.

[20] M. N. Huhns, M. P. Singh, Service-Oriented Computing: Key concepts and principles, IEEE Internet Computing 9 (1) (2005) 75–81.

[21] International Organization for Standardization, Software engineering - product quality - part 1: Quality model, ISO 9126.

[22] T. Joachims, A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization, in: 14th International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, Morgan Kaufmann, 1997.

[23] R. Johnson, J2EE development frameworks, Computer 38 (1) (2005) 107–110.

[24] T. C. Jones, Estimating Software Costs, McGraw-Hill Inc., Hightstown, NJ, USA, 1998.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with ASPECTJ, Communications of the ACM 44 (10) (2001) 59–65.

[26] M.-C. Kim, K.-S. Choi, A comparison of collocation-based similarity measures in query expansion, Information Processing & Management 35 (1) (1999) 19–30.

[27] R. Kittredge, Sublanguages, American Journal of Computational Linguistics 8 (2) (1982) 79–84.

[28] R. R. Korfhage, Information Storage and Retrieval, John Wiley & Sons, Inc., New York, NY, USA, 1997.

[29] A. Kozlenkov, G. Spanoudakis, A. Zisman, V. Fasoulas, F. S. Cid, Architecture-driven service discovery for service centric systems, International Journal of Web Services research 4 (2) (2007) 82–113.

[30] V. Lakshmi Narasimhan, B. Hendradjaya, Some theoretical considerations for a suite of metrics for the integration of software components, Information Sciences 17 (3) (2007) 844–864.

[31] V. Lertnattee, T. Theeramunkong, Effect of term distributions on centroid-based text categorization, Information Sciences 158 (2004) 89–115.

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

[32] R. M. Losee, Sublanguage terms: Dictionaries, usage, and automatic classification, Journal of the American Society for Information Science 46 (7) (1995) 519–529.

[33] R. C. Martin, Object-Oriented Design Quality Metrics: An Analysis of Dependencies, Report on Object Analysis and Design 2 (3).

[34] C. Mateos, M. Crasso, A. Zunino, M. Campo, Supporting ontology-based semantic matching of Web Services in MoviLog, in: Advances in Artificial Intelligence, 2nd International Joint Conference: 10th Ibero-American Conference on AI, 18th Brazilian AI Symposium (IBERAMIA-SBIA 2006), vol. 4140 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2006.

[35] R. McCool, Rethinking the Semantic Web. Part I, IEEE Internet Computing 9 (6) (2005) 88, 86–87.

[36] S. A. McIlraith, D. L. Martin, Bringing Semantics to Web Services, IEEE Intelligent Systems 18 (1) (2003) 90–93.

[37] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, F. Casati, Semi-automated adaptation of service interactions, in: 16th international conference on World Wide Web (WWW '07), Banff, Alberta, Canada, ACM Press, New York, NY, USA, 2007.

[38] OASIS Consortium, UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, `http://uddi.org/pubs/uddi_v3.htm` (Oct. 2004).

[39] M. Paolucci, K. Sycara, Autonomous semantic Web Services, IEEE Internet Computing 7 (5) (2003) 34–41.

[40] M. P. Papazoglou, W.-J. Heuvel, Service Oriented Architectures: Approaches, Technologies and Research Issues, The VLDB Journal 16 (3) (2007) 389–415.

[41] M. F. Porter, An algorithm for suffix stripping, Readings in Information Retrieval (1997) 313–316.

[42] S. Ran, A model for Web Services discovery with QoS, SIGecom Exchanges 4 (1) (2003) 1–10.

[43] E. Razina, D. Janzen, Effects of Dependency Injection on Maintainability, in: 11th IASTED International Conference on Software Engineering and Applications (SEA '07), Cambridge, MA, USA, ACTA Press, Calgary, AB, Canada, 2007.

[44] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, A. R. Hurson, TF-ICF: A new term weighting scheme for clustering dynamic data streams, in: 5th International Conference on Machine Learning and Applications (ICMLA '06), Orlando, Florida, USA, IEEE Computer Society, Washington, DC, USA, 2006.

[45] M. P. Reséndiz, J. O. O. Aguirre, Dynamic invocation of Web Services by using aspect-oriented programming, 2nd International Conference on Electrical and Electronics Engineering, Mexico City, Mexico (2005) 48–51.

[46] G. Salton, A.Wong, C. S. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (11) (1975) 613–620.

EasySOC: Making Web Service Outsourcing Easier. (M. Crasso, C. Mateos, A. Zunino, M. Campo). Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering. Elsevier Science. ISSN: 0020-0255. Ed.: Marek Reformat. 2010. Article in Press. DOI: doi:10.1016/j.ins.2010.01.013

[47] G. Salton, C. Buckley, Term-weighting approaches in automatic text retrieval, Information Processing & Management 24 (5) (1988) 513–523.

[48] C. Schmidt, M. Parashar, A peer-to-peer approach to Web Service discovery, World Wide Web 7 (2) (2004) 211–229.

[49] N. Shadbolt, T. Berners-Lee, W. Hall, The semantic web revisited, IEEE Intelligent Systems 21 (3) (2006) 96–101.

[50] M. Shamsfard, A. A. Barforoush, Learning ontologies from natural language texts, International Journal of Human-Computer Studies 60 (2004) 17–63.

[51] K. Sivashanmugam, K. Verma, A. P. Sheth, J. A. Miller, Adding semantics to Web Services standards, in: L.-J. Zhang (ed.), 2003 International Conference on Web Services (ICWS'03), Las Vegas, NV, USA, CSREA Press, 2003.

[52] D. Spinellis, The way we program, IEEE Software 25 (4) (2008) 89–91.

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

[54] D. Suvée, W. Vanderperren, V. Jonckers, Jasco: an aspect-oriented approach tailored for component based software development, in: 2nd International Conference on Aspect-oriented Software Development (AOSD '03), Boston, MA, USA, ACM Press, New York, NY, USA, 2003.

[55] S. J. Vaughan-Nichols, Web Services: Beyond the hype, Computer 35 (2) (2002) 18–21.

[56] S. Vinoski, A time for reflection [software reflection], Internet Computing 9 (1) (2005) 86–89.

[57] P. Vitharana, H. Jain, F. Zahedi, Strategy-based design of reusable business components, IEEE Transactions on Systems, Man, and Cybernetics 34 (4) (2004) 460–474.

[58] W3C Consortium, WSDL Version 2.0 Part 1: Core Language, W3C Candidate Recommendation, `http://www.w3.org/TR/wsdl20` (Jun. 2007).

[59] H. Wang, J. Z. Huang, Y. Qu, J. Xie, Web Services: Problems and Future Directions, Journal of Web Semantics 1 (3) (2004) 309–320.