SADIO Electronic Journal of Informatics and Operations Research

http://www.sadio.org.ar

vol. 10, no. 1, pp. 92-107 (2011)

An Evaluation on Developer's Acceptance of EasySOC: A Development Model for Service-Oriented Computing[?]

Cristian Mateos[1] Marco Crasso[1] Alejandro Zunino[1] Marcelo Campo[1]

[1] ISISTAN Research Institute.
Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN)
Campus Universitario, Tandil (B7001BBO)
Buenos Aires, Argentina
Also CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
e-mail: cmateos@conicet.gov.ar
Tel. +54-2293-440363, ext. 35

Abstract

Different development methods have been proposed for enabling the construction of software with the Service-Oriented Computing (SOC) paradigm. Service-oriented applications invoke services that developers must first discover, engage, and in time potentially replace with newer versions or even alternative services from different providers. Hence, the widely agreed development methodology involves three main activities, including service discovery, service incorporation into applications, and service replacement. EasySOC is a very recent approach for developing service-oriented applications that decreases the costs of building this kind of applications, by simplifying discovery, integration and replacement of services. This paper reports some experiments evidencing the effort needed to start producing service–oriented applications with EasySOC. Results show that users non experienced in SOC development perceive that EasySOC is convenient and easy to adopt.

Keywords: Service-Oriented Computing; Contract-last Service Consumption; Development Models; Developers' Acceptance; Start-up Curve For Building Service-Oriented Applications

1 Introduction

The success encountered by the Internet encourages practitioners, companies and governments to create software that uses information and services that third-parties have made public in the Web. Service-Oriented Computing (SOC) is a new paradigm that supports the development of distributed applications in heterogeneous environments (Erickson & Siau, 2008). SOC is a way of structuring third-party software components, which are offered as publicly available services, to accomplish a number of functional requirements. This naturally allows for a multiplicity of definitions of SOC since many relatively similar arrangements of services are possible. However, the general consensus from most available definitions is that there are three starring players within the SOC paradigm: a service provider, a service consumer and a service registry (Heuvel, 2006). Providers are entities such

[?] This article is an extended version of the paper presented in the 11th Argentine Symposium on Software Engineering (ASSE2010) - 39th JAIIO as practitioners, companies or governmental agencies that expose services. Consumers are other entities looking for such services to integrate them into their applications. The point of the service registry is for providers to advertise their services, so that consumers looking for such services can easily locate and use them. In this context, a service is a software component offered by a provider through a publicly available interface, or "technical contract" (Erl, 2007). The terms "service interface" and "service contract" will be used interchangeably in the paper.

For those people who have some background on component-based software, a good starting point to understand SOC is to view a service-oriented application as a component-based application that is created by assembling together two types of components: internal, which are those locally embedded into the application, and external, which are those bound to existing third-party services. Many enterprises in their early use of SOC assume that they can servify [1] existing applications just by concealing the details for remotely invoking services (interaction protocols, data-type formats, and distribution) behind local service proxies. Manually looking for a service contract that offers a desired functionality, interpreting its contract to generate client-side code for representing the remote service, and adapting internal components to make them compatible with the service interface is by now commonplace in the software industry. This methodology for developing service-oriented applications is known as "contract-first" (Mateos, Crasso, Zunino, & Campo, 2010).

The main trade-off of the contract-first methodology is that it allows separating business logic from technical aspects related to remote service calls, but it still fails at isolating internal components from the interfaces of the services. This is because those internal components depending on a service are tightly coupled to the interface prescribed by the service provider. In other words, client applications end up tied to the signatures of the operations of a service and therefore the data model exposed by its provider. This problem also arises, for example, in conventional GUI applications that use a certain graphic library whose API may differ from similar libraries. Then, internal components that are subordinated to particular service interfaces have to be modified and/or re-tested every time providers perform changes. 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 contract expose the same public interface (Cavallaro & Di Nitto, 2008), or that service interfaces do not suffer modifications. Therefore, as service replacement may be a recurrent situation, contract-first applications result difficult to modify and test.

During 2009, a new approach to develop service-oriented applications has been proposed. The novelty introduced by this approach is that internal components must adhere to abstract interfaces standing for services, but developers must first specify such interfaces instead of interpreting those described in service contracts. Accordingly, abstract interfaces may differ from actual service interfaces, thus this approach separates internal components from service contracts. Then, at design time, logical interactions among internal components and services are modeled through abstract interfaces without considering neither distribution and communication concerns nor third-party contracts. Conceptually, this is done by introducing an intermediate software layer that adapts abstract interfaces to actual service interfaces, isolating application components from the details of specific services. Since the development of abstract interfaces and the internal components that interact with them can take place before discovering third-party services, this approach is called "contract-last" or "code-first" (Crasso, Mateos, Zunino, & Campo, 2010b). In this context, "code" refers to the source code artifacts implementing the behavior of internal components and the interfaces of external ones.

The code-first approach for incorporating external services into applications has shown that achieves better component decoupling than contract-first, by reducing efferent couplings among internal and external components (Mateos, Crasso, Zunino, & Campo, 2010). Unfortunately, when software engineers decide to adopt the code-first approach, developers find that most existing frameworks in this direction are based on ad-hoc techniques that force them to put additional efforts to achieve mastery in these techniques. Another obstacle that hinders the adoption of code-first is that, traditionally, SOC development has heavily relied on contract-first frameworks, such as the WSIF (Duftler, Mukhi, Slominski, & Weerawarana, 2001), the Apache CXF [2], Spring [3] and Eclipse WTP [4]. Therefore, building truly loose coupled service-oriented applications using the code-first approach imposes a radical shift in the way applications are developed by the software industry. This means that a company willing to employ code-

[1] To incorporate services into applications.
[2] CFX http://cxf.apache.org
[3] Spring http://www.springsource.org
[4] Eclipse WTP http://www.eclipse.org/webtools

first methods to start producing service-oriented applications, or servify some pieces of an already existing product, would have to invest much time in training its development team, which results in a costly start-up curve.

EasySOC (Crasso, Mateos, Zunino, & Campo, 2010a) is a model for constructing service-oriented applications that encourages developers to firstly design, implement and test the internal components of an application, and focus on the servification of the application afterward. Servification means finding and incorporating into an application the external services it needs to deliver its intended functionality. Moreover, EasySOC advocates the code-first approach. One main difference among EasySOC and others code-first approaches is that it uses pervasive design patterns to establish looser relationships between internal components and service contracts. With EasySOC, the Adapter design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) enables internal components to seamlessly operate with different contracts by altering certain adapters that are responsible for dealing with the adaptation concern. The Dependency Injection (DI) (Yang, Tempero, & Melton, 2008) design pattern is employed for assembling internal components and adapters. Therefore, replacing any service involves disassembling the internal components with the old adapter, building a new adapter and assembling these components with it, while the internal components remain untouched.

EasySOC comes with a tool-box specially designed for accompanying developers during the life-cycle of their SOC applications (Crasso, Mateos, Zunino, & Campo, 2010a) (Crasso, Mateos, Zunino, & Campo, 2010b). The tool-box performs some development tasks on behalf of the users. Among these tasks it is worth remarking the generation of effective queries for performing service discovery, the adaptation between expected interfaces and actual services, the assembly of internal components depending on services, and the replacement of a service. Consequently, the EasySOC tool frees developers from dealing with technological details for discovering, invoking and assembling services, such as reaching a registry, preparing queries, interpreting search results, building proxies, and finally adapting and injecting them into target applications.

In order to empirically assess the implications of using the EasySOC development model and its tool-box both have been measured in terms of the effort needed to discover external services, and memory and CPU overheads introduced by service adapters. Further details about the experiments are reported in (Crasso, Mateos, Zunino, & Campo, 2010b). This paper represents a step towards assessing the impact of EasySOC on the software development process itself from an engineering point of view. Concretely, we performed further experiments to test the following hypothesis: understanding pervasive design patterns (i.e. Adapter and DI) and the philosophy behind code-first are the only required intellectual activities to start developing service-oriented applications with EasySOC, which should sharpen the learning curve needed to develop truly loose coupled service-oriented applications. The hypothesis has been tested with 45 postgraduate and undergraduate students of the Systems Engineering program at the UNICEN during 2009 by conducting a number of controlled development exercises. Results showed that the participants perceived that the proposed approach is convenient and thus may be easily adopted.

Basing on the fact that the surveyed students had very good programming skills but not much background on service orientation prior to the experiment, and assuming that this is the initial state of development teams planning to adopt the SOC paradigm, results suggest that EasySOC may speed up the development of service-oriented applications in real-world software factories. This also assumes that developers are accustomed to employ design patterns, which holds as today's enterprise frameworks and IDEs commonly enforce the usage of patterns such as Adapter and DI (Yang, Tempero, & Melton, 2008). Finally, although the experiments were carried out by using an IDE-specific EasySOC tool-box, our approach is completely technology agnostic, as we will explain later. In this sense, the concepts underpinning EasySOC can be materialized in a variety of programming languages and IDEs.

The rest of the paper is organized as follows. Section 2 Related work surveys approaches for developing service-oriented applications. Subsequently, Section 3 The EasySOC development model explains how our proposal improves over them, and Section 4 Supporting tool-box briefly presents its tool-box. The experimental evaluation is shown in Section 5 Experiments, and finally Section 6 Conclusions and future work presents conclusions and future work .

2 Related work

There are two main approaches for incorporating external services into service-oriented applications. The most popular approach is implemented by common software industry frameworks that mislead developers to the path of coupling specific interfaces in their service-oriented applications, thus the business logic is affected by changes in service interfaces. Instead, other efforts aimed at providing programming models to further isolate applications from services by exploiting the principle of separation of concerns. WSSI (Reséndiz & Olmedo, 2005) uses aspect-oriented techniques to dynamically replace a method with a similar operation offered by an external service. WSSI aims to automatically discover and adapt services at run-time, which has been criticized since it is arguably difficult to incorporate an appropriate service into an application without any human intervention.

In this sense, similar but semi-automatic tools have been proposed. WSML (Cibrán, Verheecke, Vanderperren, Suvée, & Jonckers, 2007) employs an aspect-oriented language, named JAsCo, to intercept and adapt client requests to actual service contracts, based on user-provided code in JAsCo. Other two works (Nezhad, Benatallah, Martens, Curbera, & Casati, 2007) (Cavallaro & Di Nitto, 2008) propose to semi-automatically generate service representatives that adapt client-side and third-party contracts. Conceptually, these two approaches require developers to specify how the expected interfaces look like and how a certain pair of expected and actual interfaces should be aligned. To this end, in (Nezhad, Benatallah, Martens, Curbera, & Casati, 2007) representatives include framework placeholders in which the programmer can manually specify the code needed to resolve ambiguities, which requires knowledge on the framework. In (Cavallaro & Di Nitto, 2008), such specifications are stated by using a custom XML language of common mapping functions. This idea is refined in (Nagano, Hasegawa, Ohsuga, & Honiden, 2004) by making such specifications more generic and associating static stubs with them. By doing so, the same stub can bind to several services. However, the generality required by the client-side specifications comes at the expense of significant domain knowledge.

The above efforts accommodate the interfaces of the services to the ones specified and required by developers at design time. These efforts are based on ad-hoc languages and programming models that are intuitively difficult to adopt. Unlike them, EasySOC combines the Adapter design pattern with Dependency Injection (DI), a popular programming style among developers. Moreover, though the authors of the mentioned efforts have meticulously positioned their approaches from a modeling perspective with respect to related research, the soundness of (Cibrán, Verheecke, Vanderperren, Suvée, & Jonckers, 2007) (Nagano, Hasegawa, Ohsuga, & Honiden, 2004) (Nezhad, Benatallah, Martens, Curbera, & Casati, 2007) has not been corroborated experimentally yet. On the other hand, the feasibility of EasySOC has been empirically evaluated and reported (Mateos, Crasso, Zunino, & Campo, 2010) (Crasso, Mateos, Zunino, & Campo, 2010b). The next section presents EasySOC in detail, while an assessment of its acceptance is presented in Section 5 Experiments.

3 The EasySOC development model

The EasySOC development model prescribes a novel and easy methodology to design service-oriented applications and guide developers during the entire life-cycle of their software. Metaphorically, central to this methodology is to think of service-oriented applications as special puzzles. Such special puzzles have two types of pieces. One type of pieces represents the internal components of a service-oriented application (the ones implemented by users), whereas another stands for third-party services (the ones not implemented but discovered and reused). Hence, service pieces have some peculiarities. First, they are public and as such they can be used to solve many puzzles, i.e. called from different applications. Second, there are many service pieces with the same content, so the puzzle solver -in this case a developer- should select among the available alternatives. Third, the shape of service pieces can be slightly modified, or adapted, to fit into a puzzle without affecting those puzzles already using them. Here, the shape represents the interface with the operations offered by a service piece. Figure 1 illustrates this metaphor. Internal component pieces and service pieces are depicted using gray and black, respectively.

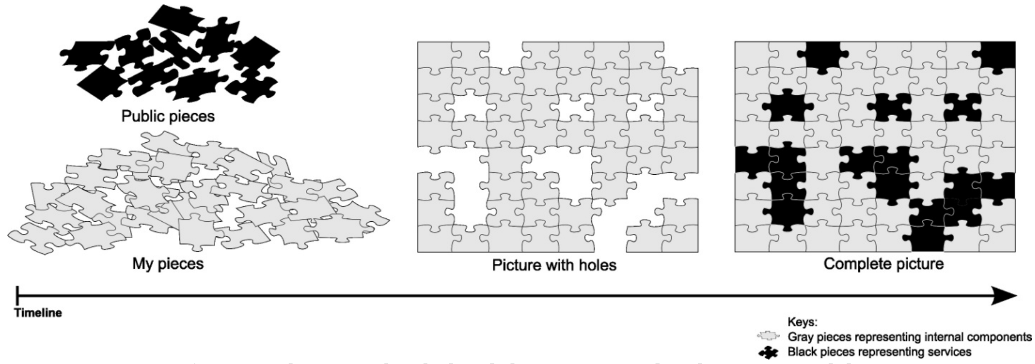

Figure 1 The metaphor behind the EasySOC development model.

The approach taken by EasySOC to build these special puzzles is to start by joining gray pieces. Once the gray parts of the puzzle are built, the solver should look for every hole in the "picture", and pick a proper public piece to fill it. This should be iteratively applied to associate a black piece with each hole in order to complete the picture. Then, when building a service-oriented application with EasySOC, a developer thinks of such an application as a collection of internal components invoking external ones, i.e. services. Having in mind SOC applications as arrangements of internal components and services, EasySOC encourages developers to first design, implement and test for correctness the internal components of their applications, and then discover and incorporate services into them. This is analogous to first arrange gray pieces together, and in turn fill the holes of the resulting picture using black ones.

There are, naturally, many similar ways of designing and implementing the pieces or components of an application. However, not necessarily all the alternatives to arrange internal components and services are viable. For instance, several researchers have shown that the alternative adopted by common libraries for invoking Web Services misleads developers to build service-oriented applications that are rather hard to understand and to maintain (Crasso, Mateos, Zunino, & Campo, 2010a). Unlike these libraries, EasySOC proposes a programming method to arrange the components and services of a SOC application that facilitates their maintenance afterward (Mateos, Crasso, Zunino, & Campo, 2010). This is achieved by raising the level of abstraction by which the necessary plumbing is done while providing a programming model that is very close to component orientation.

The method to develop service-oriented applications with EasySOC consists of two groups of tasks. The first group includes design, implementation and test of internal components, whereas the activities in the second group deal with servification of external components. Servifying with EasySOC involves three steps: (1) finding a list of candidate services, (2) selecting an individual service from this list, and (3) injecting an adapted representative of the selected service into the application.

The EasySOC three-step servification method takes as input an incomplete application, where some of its constituent components are implemented, and others are intended to be delegated to services. Graphically, this kind of applications is shown in Figure 2, using the UML 2.0 notation for modeling components. Based on the dependencies between the internal and the external components of the input application, the aforementioned three steps are iteratively applied to quickly and seamlessly associate an individual service with each one of the external components. Overall, the discovery-selection-injection sequence is performed until all external components of the input application have been associated with a service. Under a service replacement scenario, steps 2 and 3 should be re-performed.

As a result of performing the three-step method, a developer thinks of a service as any other regular component providing a clear interface to its operations. If a developer wants to call an external service S with interface I_S from within internal components C´ and C´´, a dependency among these two latter and S is established through I_S. This kind of dependency is commonly managed by a DI container that injects a proxy to S (let us say P_S) into C´ and C´´. At run-time, the code of the internal components will end up calling any of the methods declared in I_S through P_S, which transparently invokes the remote service. Interestingly, this mechanism is not intrusive, since it only requires associating a configuration file with the client application, which is used by the DI container to determine which components should be injected into other ones.

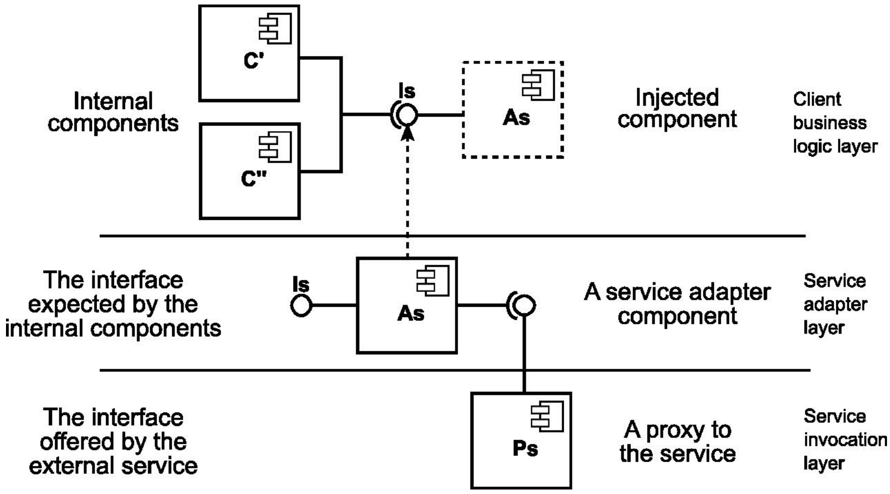

Figure 2 Anatomy of EasySOC applications.

Although DI provides a fitting alternative to cleanly incorporate a service into an application, it leads to a form of coupling through which the application is tied to the invoked service contracts (i.e. I_S). In this way, changing the provider for a service requires to adapt the client application to follow the new contract. To overcome this problem, EasySOC takes DI a further step and combines it with the Adapter pattern to introduce an intermediate layer that allows developers to seamlessly shift between different contracts.

Conceptually, instead of directly injecting a layer of service proxies (P_S) into the application, which requires modifying the layer containing the client business logic in such a way it is compatible with the service contracts (I_S), EasySOC injects a layer of service adapters. A service adapter is a specialized proxy, which adapts the interface of a particular service according to the abstract interface (specified by the developer at design time) expected by the internal components.

We refer to A_S as a service adapter that accommodates the actual interface of a service S to the interface expected by internal components. In other words, service adapters carry the necessary logic to transform the operation signatures of the interfaces expected by clients to the actual interfaces of selected services. For instance, if a service operation returns a list of integers, but the application expects an array of floats, a service adapter would perform the type conversion.

The next section describes the tool-box provided by EasySOC, which allows developers to perform the first and third steps of the proposed servification method automatically and semi-automatically, respectively.

4 Supporting tool-box

Despite the positive aspects of the proposed development model to decouple service consumers and providers, the solution to the problem relies on the tasks of discovering services, adapting service interfaces and assembling dependencies into dependants, which are not trivial and might involve high development costs. To overcome these costs, we have built a plug-in for the Eclipse IDE that aims at automatically performing these tasks on behalf of SOC application developers. The tool has been designed to implement the SOC paradigm using Web Service technologies (Erickson & Siau, 2008) and Java. Tutorials, screen-shots and a setup file can be downloaded from the plug-in home page5. The next subsections describe the discovery and incorporation modules of the tool-box, and a step-by-step usage example.

4.1 Service discovery

The EasySOC tool-box exploits the concept of Query-By-Example for Web Services and the approach to generate queries described in (Crasso, Zunino, & Campo, 2009). This concept suggests that because of the structure inherent to code-first applications, an abstract interface (I_S) can be seen as an example of what a developer is looking for. Consequently, the EasySOC tool-box gathers certain information that is implicitly conveyed in the source code of external component interfaces, which is preprocessed to build a refined description of developers' needs. Accordingly, an effective query is generated provided developers follow documenting and naming best practices in their service-oriented applications. This is because the query generation heuristic gathers relevant terms from the names, comments, and operations and arguments of an interface. Finally, the query is sent to a registry and returned results are presented to developers.

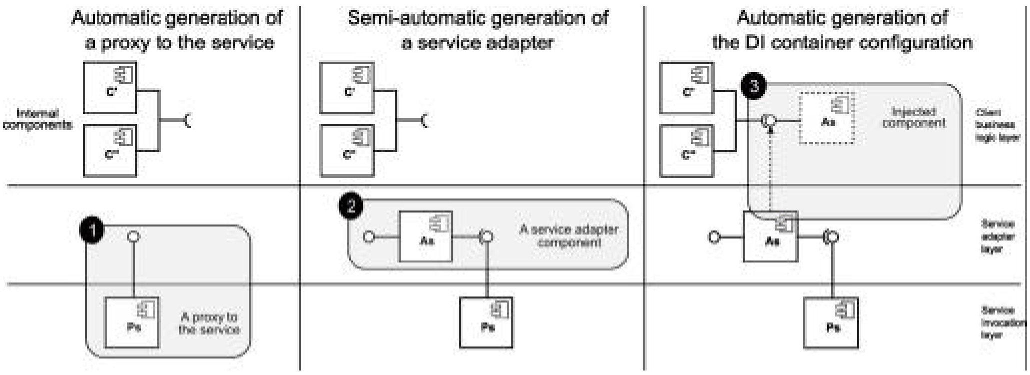

Figure 3 Development steps with EasySOC tool-box.

4.2 Service Incorporation and replacement

The EasySOC tool-box automatically carries out the adaptation and assembling tasks described early. To do this, once an external service is selected, proxy construction is automatically performed by the tool-box (see Figure 3 step 1). Then, the tool-box tries to build an adapter to map the interface of the proxy onto the abstract interface that internal components expect (see Figure 3 step 2). Finally, the tool-box indicates the DI container how to assemble internal components and service adapters together (see Figure 3 step 3).

The current implementation of the EasySOC tool-box employs Axis2 for building service proxies, and Spring as the DI Container. Building a proxy with Axis2 involves giving as input the interface description of the target service (a WSDL 6 document) to a command line tool. To setup the DI container, the names of dependants and services must be written in an XML file. For adapting external service interfaces to the internal abstract ones, we have designed an algorithm based on the work published in (Stroulia & Wang, 2005).

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

5 http://sites.google.com/site/easysoc/home/service-adapter
6 http://www.w3.org/TR/wsdl

arguments. Data-types similarity is based on a pre-defined similarity table that assigns similarity values to pairs of simple data-types. The similarity between two complex data-types is computed in a recursive way. Once a pair of operations has been chosen, service adapter code is generated. To do this, the algorithm adapts simple data-types by taking advantage of type hierarchies and performing explicit type conversions (castings). Complex data-types are resolved recursively as well. Clearly, not all available mismatches are covered by the algorithm, thus developers should revise the generated code, which makes the incorporation step semi-automatic.

4.3 Using EasySOC: Step-by-step example

To understand the implications of modeling SOC applications with EasySOC, this section describes the design of a service-based personal agenda. The personal agenda is in charge of managing a contact list, arranging new meetings, and to notify these contacts of new planned meetings. The contact list is modeled as a collection of records with information about individuals such as name, current address (city, state, country, zip code, etc.), telephones, email addresses, etc. For the sake of clarity, we have simplified the functionality for coordinating the meeting by assuming that the participants being notified always agree with the arrangement provided by the requesting user.

Below we list the activities carried out by the personal agenda upon the creation of a new meeting. The text in italics represents the functionalities that will be not implemented but delegated to Web Services. We assume that the user of the personal agenda provides the date, time, participants and location of the meeting upon its arrangement. Algorithmically, creating a new meeting roughly involves:

1. *Getting a weather forecast* for the meeting place at the desired date and time.
2. *Obtaining the routes* (or driving directions) that each contact participating in the meeting could employ to travel from their current address to the meeting place.
3. For each participant of the meeting:

    a. Creating an email with an appropriate subject, and a body including the weather report and the obtained route information.
    b. *Spell checking* the text of the email.
    c. *Sending the email*.

To build the above application, we start by designing its internal components. First, we define an internal component called PersonalAgenda, which is at the heart of the application and is in charge of coordinating the various services necessary for arranging a new meeting, and a ContactManager component representing the contact list. Then, we define four abstract interfaces used by the PersonalAgenda component, namely:

- IForecast: returns a weather report for a given ZIP code.
- IRouteInfo: supplies driving directions for a given source and target locations.
- ISpellChecking: detects spelling mistakes in a given text.
- IEmailSending: sends an email using a given body text and email address.

At this point, our application consists of two internal components and four abstract interfaces. Then, we employ the EasySOC tool support for discovering services that provide a concrete implementation for the functionality modeled by these abstract interfaces. For instance, the next abstract interface Java code is used as a query when looking for spell checking services:

```
/**
 * Spell checks a piece of text. Wrong words are replaced
 * in the result by their correct counterpart.
 *
 * @param text Text to spell check.
 * @result The automatically spell checked text.
 */
```

```
public interface ISpellChecking {
        public String spellCheck(String text);
}
```

Figure 4 depicts the GUI of our plug-in within the Eclipse IDE. When discovering and then associating concrete Web Services instances with an abstract interface, users simply indicate through a dialog such interface. The dialog shows how many services and categories are available in the registry about to be queried. Users are allowed to perform advanced searches, for example looking for services within an individual category. Finally, after querying the registry, a candidate list is presented to the user, as shown in Figure 5 (bottom). For each candidate Web Service, the offered operations are shown. Users can further browse and visualize the arguments and results of each operation in Java as well as WSDL format.

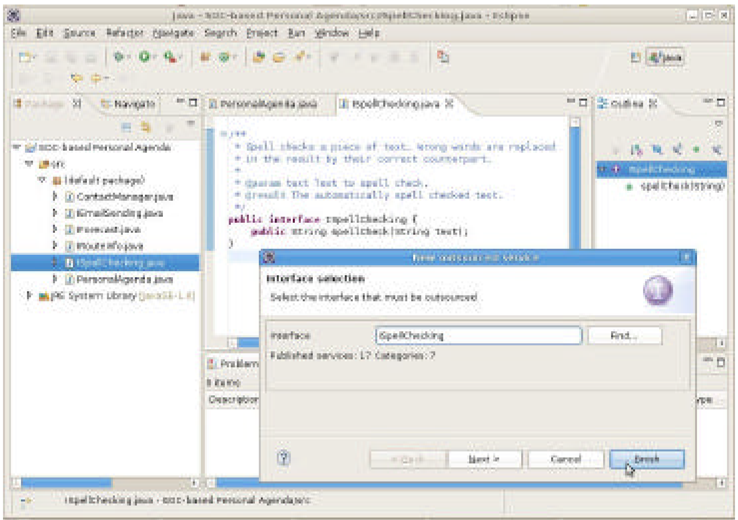

Figure 4 The EasySOC discovery support in action.

In our example, we have obtained two candidates services labeled "spellchecker" and a more specific "englishspellchecker". Let us suppose we select the second candidate, thus we must command our tool to build the corresponding proxy and adapter, and assemble them to the PersonalAgenda component. This is done via a contextual menu from the selected service (see Figure 5). Afterwards, we revise the generated adapter code to ensure that all signature and data-type mappings are properly coded and specified. The generated extra code artifacts will appear in a separate "mappings" folder of the project in the left part of the GUI.

Overall, the discovery-selection-injection sequence is performed until all external components of the application are associated with a service. To conclude, it is worth noting that user intervention was only required to select candidate services and revise adapters. Since an adapter indirectly interacts with a service through a proxy using the built-in object-oriented mechanism of method invocation, the user was free from dealing with WSDL, SOAP and other SOC-specific technologies in the code of the application.

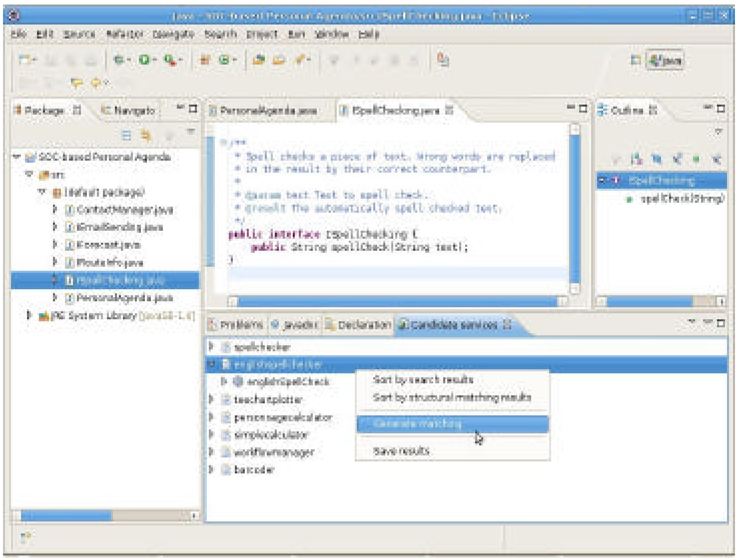

Figure 5 EasySOC service selection view.

5 Experiments

This section introduces the experiments that were performed in order to assess whether the EasySOC development model has an acceptable difficulty of adoption by novice developers. The experiments involved 45 students and a two-phase homework, after which the students were asked to complete a survey to collect their opinions about the whole experience. The work was carried out individually by the students, and each part of the work impacted on the partial and final grades for the course. This contributed to ensure a high level of commitment with the evaluation. As the experiment involved the use of a tool-box of our own, which might represent a threat to validity, the students were not tell about the secondary goal of the homework, and precise and careful instructions prior to take the survey were emailed to them to ensure objectivity.

The experiments were in the context of the "Service-Oriented Computing"[7] course of the Systems Engineering at the Faculty of Exact Sciences (Department of Computer Science) of the UNICEN during 2009. The course was also offered in 2008, is optional, and its audience are last-year undergraduate students and postgraduate students (both master and doctoral programs) without knowledge on SOC concepts. The course requirements are excellent skills on programming and some experience with Java development. In 2009, the course was taken by 38 undergraduate students and 7 postgraduate students from four different Universities.

After five lectures within one week of three hours each discussing the fundamentals of the SOC paradigm and enabling technologies the students were instructed to develop a service-based personal agenda software by outsourcing some Web Services from a registry given as an input. The course content comprises traditional technologies, such as WSDL, SOAP, Eclipse WTP, WSIF, but also EasySOC. Basically, the main responsibilities of the personal agenda software were to manage a user's contact list and to notify these contacts of events related to planned meetings. The contact list was a collection of records, where each record keeps information about an individual such as name, location, email, and so on. The students were also given a pseudo-algorithm of the functionality for arranging meetings, and some hints on which components of the agenda software could be implemented as Web Services.

The development of the software involved two phases. The second assignment was given after finishing the first one. In the first phase, the students implemented the agenda software by using traditional Web Service technologies from the set of alternatives discussed in the course lectures. Basically, the technologies were needed to inspect the service registry and to consume and incorporate selected services into the software. In the second phase, the students developed the same software by using EasySOC. Therefore, in principle, the first phase required an initial

7 http://www.exa.unicen.edu.ar/~cmateos/cos

exploratory research in order to come out with the technologies to be used, whereas the second phase involved the use of EasySOC and as such did not required much effort in this respect. The assignments were developed based on the Eclipse IDE. In both phases, the students exercised three aspects inherent to developing SOC applications, namely:

1. *Service discovery*: In the first phase, this was carried out by inspecting the input service registry through a "Google-like" GUI that supported keyword-based search of Web Services. In the second phase, this was performed by using the Web Service discovery support of EasySOC.

2. *Service incorporation*: In the first phase, this involved building service proxies based on the service invocation capabilities of the Web Service technology individually chosen by each student, whereas in the second phase this was uniformly handled by using the incorporation facilities of EasySOC.

3. *Service replacement*: The input service registry had several implementations for the Web Services needed to develop the agenda software. The students were asked to change the provider for a half of the services *after* implementing their software. For both phases, this involved repeatedly performing (1) followed by (2) on the already implemented agenda.

To better prepare the students to fill out the survey, we added some general "warming up" questions at the beginning of the survey, asking for example what SOC is and what kind of applications actually benefits from it. Then, we included several query items designed to collect the students' opinions with respect to the three aspects mentioned above. By following Likert's approach to build questionnaires (Likert, 1932), the items were not plain questions but statements to which the students could either totally agree, agree, somewhat agree, somewhat disagree, disagree or totally disagree. In this sense, the students did not felt evaluated but consulted. We employed an even-numbered scale of agreement to better capture the students' opinions (no neutral mid-point). Additionally, they had to provide a textual justification for each item. We also reserved a check box to indicate the perceived overall difficulty of the course and its assignments, and a text field through which any further comments could be specified.

Given the different formation levels of the students involved in the experiments, the next two subsections will analyze the results by considering the opinions of the postgraduate students (PGS) and undergraduate students (UGS), respectively. Table 1 summarizes the survey query items (warming up questions have been omitted) and results. Query items were arranged in two groups, i.e. those asking whether students would use either approaches for developing service-oriented applications (items 1-2), and those evaluating the suitability of the EasySOC model according to the aspects that are inherent to SOC development from a software engineering perspective (items 3-6).

Query item	**Totally agree**	**Agree**	**Somewhat agree**	**Somewhat disagree**	**Disagree**	**Totally disagree**
I would always develop any SOC application as in the 1st phase	UGS=1 (3%)	UGS=5 (13%)	UGS=18 (47%)	UGS=7 (18%)	UGS=6 (16%)	UGS=1 (3%)
	PGS=0 (0%)	PGS=1 (14%)	PGS=2 (29%)	PGS=1 (14%)	PGS=2 (29%)	PGS=1 (14%)
I would always develop any SOC application as in the 2nd phase	UGS=1 (3%)	UGS=16 (42%)	UGS=15 (39%)	UGS=3 (8%)	UGS=2 (5%)	UGS=1 (3%)
	PGS=0 (0%)	PGS=5 (71%)	PGS=1 (14%)	PGS=0 (0%)	PGS=1 (14%)	PGS=0 (0%)
EasySOC materializes the triad SOC model	UGS=1 (3%)	UGS=9 (24%)	UGS=14 (37%)	UGS=3 (8%)	UGS=0 (0%)	UGS=2 (5%)
	PGS=3 (43%)	PGS=4 (57%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)
EasySOC abstracts from Web Service technologies	UGS=1 (3%)	UGS=14 (37%)	UGS=6 (16%)	UGS=1 (3%)	UGS=0 (0%)	UGS=0 (0%)
	PGS=5 (71%)	PGS=2 (28%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)
EasySOC simplifies service discovery	UGS=27 (71%)	UGS=9 (24%)	UGS=1 (3%)	UGS=1 (3%)	UGS=0 (0%)	UGS=0 (0%)
	PGS=5 (71%)	PGS=2 (28%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)
EasySOC helps in changing service providers	UGS=18 (47%)	UGS=11 (29%)	UGS=8 (21%)	UGS=1 (3%)	UGS=0 (0%)	UGS=0 (0%)
	PGS=6 (86%)	PGS=1 (14%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)	PGS=0 (0%)

Table 1 Results based on 38 undergraduate students (UGS) and 7 postgraduate students (PGS).

5.1 Postgraduate students: Survey analysis

For the first group of items, none of the surveyed postgraduate students completely agreed to rely on any of the two approaches for developing their service-oriented applications, as shown in Table 1. However, 85% of the students either agreed or somewhat agreed to the idea of "using EasySOC in early stages of development", since the pattern-based programming model of EasySOC could lead to some adaptation effort when servifying existing applications in order to made them compliant to the EasySOC application anatomy. However, the same students said that they would definitively use the tool in the presence of large service registries whose functional content is not known regardless the development stage. This is precisely the case of open contemporary massively distributed environments such as the Web or Grids, in which thousands of services are offered and therefore it is crucial to have effective and efficient discovery mechanisms to dramatically narrow down the result list when looking for required services (Crasso, Zunino, & Campo, 2009). Furthermore, one student disagreed with always using EasySOC because he/she through that our discovery mechanism would not be effective when dealing with poorly described WSDL documents (the same student consistently disagree with not employing any other invocation library in those cases when many services are available). This is certainly a correct observation, on which we have been in fact working on by identifying common anti-patterns in WSDL descriptions that harms our service discovery engine and providing guidelines to avoid them (Rodriguez, Crasso, Zunino, & Campo, 2010). We are therefore planning to incorporate these ideas into EasySOC tool-box in the near future.

Moreover, 4 out of the 7 students disagreed with different confidence levels to using the Web Service libraries employed in the first phase of the assignment because such libraries demanded them to significantly rewrite the application upon changing service providers. In other words, they thought that having an adaptation layer for isolating code from service interfaces is beneficial and better supports the maintainability and evolution of developed client-side software. As a complement, the other 3 students said that they would rely on the first approach to service consumption as long as the set of services to be consumed are known in advance, i.e. services are given as input to the development process. However, these 3 students consistently responded that they would switch to EasySOC in cases when target services are not determined beforehand, as some support for service discovery would then be strongly necessary.

On the other hand, for the second group of items, all postgraduate students either totally agreed or agreed to the associated query items. Most of them said that EasySOC provides intuitive support to the triad find-consume-publish of the Web Service model, even when they did not exercised the last activity in the homework but nevertheless acknowledged that the tool-box has support for it. Certainly, materializing such model directly in the development tool allows users to focus on performing the activities that correspond to their roles, i.e. service consumer or service provider. Moreover, students considered that EasySOC allowed them to be unaware of the technological details for finding or consuming services. Concretely, half of the students conceived providing code for inspecting service registries and processing WSDL Web Service descriptions as being two of the most time-consuming and tedious tasks when building their SOC applications. One student pointed out, however, that even when abstraction from technological details is important, so is to have background on low-level technologies for those cases in which specific adjustments must be made to an application (e.g. changing the communication protocol to talk to incorporated services). In this sense, EasySOC automatically generates the necessary technology-dependent software artifacts for calling external Web Services, while allows users to modify these artifacts as needed.

The seven postgraduate students found the service discovery module of EasySOC "very helpful to quickly find required candidate services", which essentially means that looking for Web Services implementing the functionality a client application expects is effective and efficient and hence has a positive impact on application building in terms of development time. Furthermore, 4 out of the 7 students found that good code documentation in their client-side software artifacts was a prerequisite for the discovery process of EasySOC to be effective. Indeed, the effectiveness in finding required services heavily depends on to what extent users employ explanatory names and proper documentation for both class names and method parameters. However, note that this does not represent a strong assumption from our tool as these are desirable and frequent (Spinellis, 2008) development practices for any kind of software. Finally, all of the students said that EasySOC helped them with the requirement of changing service providers.

5.2 Undergraduate students: Survey analysis

Table 1 shows that, for the case of undergraduate students, the opinions with respect to items 1 and 2, and to a lesser extent for the items 3-6, were less concentrated as opposed to the results of the previous subsection. In this sense, to better analyze the responses, we quantified and categorized whether each individual student was more convinced of using an approach above the other. For example, if a student agreed to "I would always develop any SOC application as in the first phase" and somewhat agreed to "I would always develop any SOC application as in the second phase", it meant that the student preferred the contract-first approach. Figure 6 illustrates the obtained results. It is worth pointing out that, except for the case of the "Undecided" group, the rest of the students either somewhat agreed, agreed or totally agreed to one of these two items, which established a minimum acceptable level of confidence regarding tool preference.

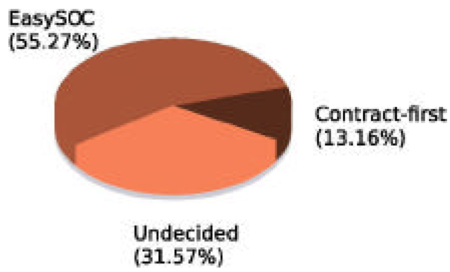

Figure 6 Undergraduate students: Approach preference.

Remarkably, 55.27% of the surveyed students said that they preferred using EasySOC over relying on tools based on contract-first. The common argument behind this preference was that the basic elements of the EasySOC programming model facilitate the "agile" development of "modifiable" SOC applications. Regarding the functionality offered by our tool-box, the students also emphasized on the usefulness of its discovery mechanisms, and the convenience of its automatic source code generation techniques, for example for building service adapters. Furthermore, 5 out of the 38 students (13.16%) said that they were more comfortable with contract-first since it required less software for calling services compared to EasySOC (just a service invocation framework), and "one could also achieve an acceptable level of decoupling between applications and service contracts by addressing this non-functional requirement early in the design stage of the application". Precisely, EasySOC comes with a software support that prescribes a simple programming model based on pervasive patterns, which leads to a natural way of building SOC applications with high levels of decoupling. Application design is thus more focused on specifying the functionality of the internal application components and the external services, while decoupling is addressed implicitly when materializing these components via our tool-box.

Not surprisingly, 31.57% of the undergraduate students were not decided about which approach they would use to develop SOC applications in the future. Moreover, half of them (i.e. 6 students) simultaneously somewhat agreed to using both tools because "choosing a development tool depends on several factors", including the size of the client-side software, the number of services to be consumed, and the amount of dependencies between internal application components and such services, or even management-level directions. However, the same students pointed out that they found EasySOC useful to simplify service discovery, and to keep the client source code away from "service-specific instructions", or in other words contract-related code.

On the other hand, the other half of the students gave origin to two corner cases. Three students agreed to employ either approaches since they had trouble learning Eclipse but they would definitively exploit the design principles materialized by EasySOC for building their applications. As stated before, these principles are technology-agnostic, and we are in fact working on providing alternative materializations of EasySOC for supporting other popular DI containers and IDEs to further ease its adoption. Lastly, two students and one student simultaneously disagree and completely disagree, respectively, on using either models for developing applications. After carefully looking at their opinions, the reason of the low level of agreements was that of the above 6 students, i.e. they were not sure about which approach would be the best option in most scenarios. One of the three students additionally pointed out that in larger projects the adapter layer injected by EasySOC might negatively affect the performance of applications compared to those not relying on adapters. After finishing collecting the students' opinions, we conducted an empirical study that showed that supporting adapters in practice has an acceptable overhead in terms of CPU and memory consumption (Mateos, Crasso, Zunino, & Campo, 2010).

5.3 Students' acceptance analysis

Finally, the well-known Likert scale (Likert, 1932), the most widely used psychometric scale in survey research, was assessed. Roughly, the Likert scale is the sum of answers on several Likert items, i.e. individual statements to which respondents can associate a level of agreement. After the survey is comp leted, the agreement levels of each Likert item are typically summed to create an overall score per participant.

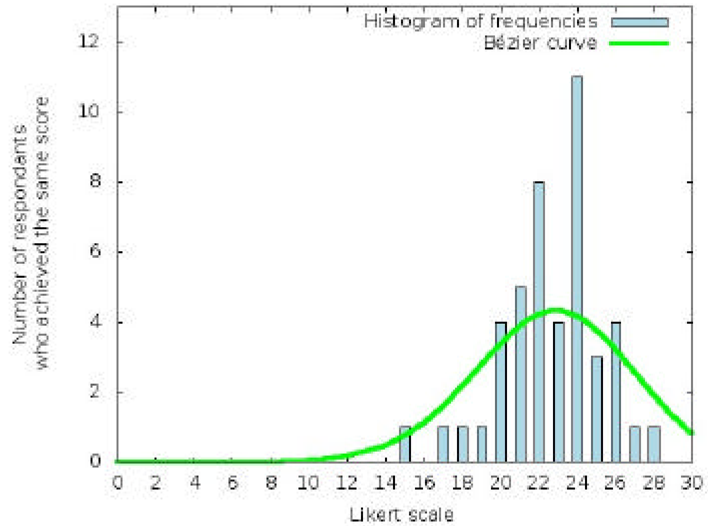

Figure 7 Likert scale: Results data distribution.

Since we were interested in quantifying the overall perception of the students on EasySOC, we associated a numerical score with query item 1 ranging from 0 (totally agree) to 5 (totally disagree), but ranging from 5 (totally agree) to 0 (totally disagree) for query items 2-6. As a consequence, our designed Likert scale was in the range of [0,30], with 0 being strongly disagree with EasySOC and with 30 being strongly agree with it. We calculated the Likert score per student. Figure 7 depicts the results frequency and a smooth curve. Frequency was calculated as the number of students who had the same score. Interestingly, only one participant got the lowest score that was 15, i.e. the worst perception was "neutral". After smoothing the results using Bézier curves, they tended to a normal distribution with an average $\mu = 22.67$ and a standard deviation $s = 2.65$, meaning that 95.4% of the students scored between $[\mu - 2 * s, \mu + 2 * s]$. In other words, 42 students scored in the range $[17.37, 27.97]$, which manifests a very good perception of EasySOC.

6 Conclusions and future work

Service-Oriented Computing is a relatively new paradigm for software development that promotes the seamless reuse of existing pieces of functionality exposed by third-parties. Nowadays, the paradigm has reached a significative level of maturity and is being actively exploited in the software industry by means of specialized frameworks for both exposing and consuming services. Particularly, tools in the latter category are based on a contract-first approach to service consumption, which commonly leads to applications that heavily depend on particular service contracts and therefore compromises maintainability. Moreover, these tools pay little attention to other essential aspects of SOC development, namely service discovery and replacement.

Alternatively, code-first focuses on achieving a stronger separation between application code and service contracts. Sadly, tools in this line are based on techniques that are difficult to use for average users. To address this, we have proposed EasySOC, a development model that materializes code-first concepts and enforces the usage of pervasive object-oriented design patterns as a way of structuring SOC applications. In recent works, we have empirically shown that EasySOC helps in easing service discovery (Crasso, Mateos, Zunino, & Campo, 2010b), improves source code maintainability and service replacement (Crasso, Mateos, Zunino, & Campo, 2010a), and does not incur in performance overheads at run-time (Mateos, Crasso, Zunino, & Campo, 2010). The evaluation presented in this paper offers complementary evidence about software practitioners' acceptance of the proposed approach.

We worked on the hypothesis that EasySOC sharpens the learning curve needed to build loosely coupled SOC applications provided developers have some required basic concepts, namely design patterns and the code-first method. We performed a controlled experiment and surveyed 45 last-grade and postgraduate students to collect their opinions. Results suggest that the students perceived EasySOC as a convenient and intuitive tool for implementing applications. Since the students had very good programming skills but not much knowledge on SOC before the experiment, which is in fact the initial state of real development teams planning to implement the SOC paradigm, we can reasonably extrapolate these results to support the argument that EasySOC may be useful in similar real-world situations. Despite these encouraging results, we will conduct experiments with other students and real development teams to further validate our claims.

The EasySOC approach and its supporting tool-box are in constant evolution and represent and effort towards not only facilitating SOC application development but also service development. Besides being development model, EasySOC can be conceived from a methodological perspective as a set of guidelines to build more maintainable and testable applications. In this sense, we have adapted our ideas to provide a more integral method that covers all the basic elements of the SOC paradigm, namely service clients, registries and publishers (Rodriguez, Crasso, Mateos, Zunino, & Campo, 2010). As a consequence, we are preparing a new release of the EasySOC tool-box that incorporates these recent addings. Eventually, we could reproduce the experiments described in this paper to assess the effectiveness of our tool regarding service creation and publication.

Acknowledgements

We deeply thank the students who participated in the survey for their good predisposition in the experiment, and the anonymous reviewers of ASSE'2010 for their constructive comments about the conference article. We also acknowledge the financial support provided by ANPCyT through grant PAE-PICT 2007-02311.

References

Cavallaro, L., & Di Nitto, E. (2008). An approach to adapt service requests to actual service interfaces. *30th International Conference on Software Engineering* (págs. 129-136). Leipzig, Germany: ACM.

Cibrán, M. A., Verheecke, B., Vanderperren, W., Suvée, D., & Jonckers, V. (2007). Aspect-Oriented Programming for Dynamic Web Service Selection, Integration and Management. *World Wide Web , 10* (3), 211-242.

Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2010a). EasySOC: Making Web Service Outsourcing Easier. *Information Sciences, Special Issue of Applications of Computational Intelligence and Machine Learning to Software Engineering , in press*.

Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2010b). Empirically Assessing the Impact of Dependency Injection on the Development of Web Service Applications. *Journal of Web Engineering ,9* (1), 66-94.

Crasso, M., Zunino, A., & Campo, M. (2009). Combining Query-By-Example and Query Expansion for simplifying Web Service discovery. *Information Systems Frontiers , in press*.

Duftler, M. J., Mukhi, N. K., Slominski, A., & Weerawarana, S. (2001). Web Services Invocation Framework WSIF. *OOPSLA*. Tampa Bay, Florida, USA: ACM.

Erickson, J., & Siau, K. (2008). Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating Hype from Reality. *Journal of Database Management , 19* (3), 42-54.

Erl, T. (2007). *SOA Principles of Service design.* Prentice Hall.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Heuvel, M. P.-J. (2006). Service-oriented design and development methodology. *International Journal of Web Engineering and Technology , 2* (4), 412-442.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology , 22* (140).

Mateos, C., Crasso, M., Zunino, A., & Campo, M. (2010). Separation of Concerns in Service-Oriented Applications Based on Pervasive Design Patterns. *ACM Symposium on Applied Computing* (págs. 2509-2513). Sierre, Switzerland: ACM.

Nagano, S., Hasegawa, T., Ohsuga, A., & Honiden, S. (2004). Dynamic Invocation Model of Web Services Using Subsumption Relations. (págs. 150-157). IEEE Computer Society.

Nezhad, H. M., Benatallah, B., Martens, A., Curbera, F., & Casati, F. (2007). Semi-Automated Adaptation of Service Interactions. (págs. 993-1002). ACM.

Reséndiz, M. P., & Olmedo, J. O. (2005). Dynamic invocation of Web Services by using AOP. (págs. 48-51). IEEE Computer Society.

Rodriguez, J. M., Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2010). The EasySOC Project: A Rich Catalog of Best Practices for Developing Web Service Applications. *Proceedings of the XXIX International Conference of the Chilean Computer Society (IC-SCCC 2010)*, (p. in press). Antofagasta, Chile.

Rodriguez, J. M., Crasso, M., Zunino, A., & Campo, M. (2010). Improving Web Service Descriptions for effective service discovery. *Science of Computer Programming , 75* (11), 1001-1021.

Spinellis, D. (2008). The Way We Program. *IEEE Software , 25* (4), 89-91.

Stroulia, E., & Wang, Y. (2005). Structural and Semantic Matching for Assessing Web Service Similarity. *International Journal of Cooperative Information Systems , 14* (4), 407-438.

Yang, H. Y., Tempero, E., & Melton, H. (2008). An Empirical Study into Use of Dependency Injection in Java. (págs. 239-247). IEEE Computer Society.