Separation of Concerns in Service-Oriented Applications Based on Pervasive Design Patterns

[Author(s) name(s) and address(es) were removed for blind review purposes]

ABSTRACT

Service-Oriented Computing (SOC) allows developers to build applications by reusing and invoking Web-accessible services. SOC promotes loose coupling between applications and services, which has been mostly addressed by using techniques for Separation of Concerns (SoC). Contemporary SOC development models based on SoC either rely on difficult-to-adopt, ad-hoc programming facilities and languages or fail at isolating applications from details of the application-service interaction. We propose DI4WS, a SOC programming model that combines the well-known Adapter and Dependency Injection patterns. DI4WS achieves higher levels of SoC in service-oriented applications without requiring developers to learn such facilities or languages. DI4WS follows a contract-last approach to service invocation, whereby developers first code the logic of their applications and then non-invasively "adapt" and "inject" required services. We present a formula that can be used to show that such approach allows reducing couplings to services, which has a positive effect on application maintenance. An empirical comparison of DI4WS with two related approaches is also reported, showing that the DI4WS versions of 4 evaluated applications used less memory and run faster than their counterparts.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

Keywords

Separation of concerns; Service-oriented computing; Web Services; Contract-last service consumption; Dependency Injection.

1. INTRODUCTION

Separation of Concerns (SoC) [13] is both a principle and a process for building software systems, which states that each constituent part of a system should be free of behaviors not inherent to its functional nature. Traditionally, SoC has been achieved by using modularity, encapsulation and information hiding techniques in the context of object-oriented programming, or layered designs in an architectural one. All in all, the goal of SoC is to simplify the management and maintenance of software systems by providing the guidelines for creating ordered and clean system designs.

Closely related to SoC is the notion of loose coupling, by which the components of a system know as little details as possible from each other. In essence, the idea is to allow an individual component to be altered without affecting the rest of the components, i.e. to enable black-box reuse. Component interactions are conceived as a cross-cutting concern and thus the code implementing such interactions is isolated from the code representing the pure component behavior, or core concern. This practice is fundamental for example in distributed systems, in which the code to actually support the interactions between components residing in different machines is abstracted away from the applications and is placed within a middleware instead. Indeed, component interaction is a cross-cutting concern as its code necessarily spans through the behavioral code.

One of the areas of computing that has seen an ever increasing degree of application of these notions is Service-Oriented Computing (SOC). SOC is a contemporary computing paradigm that supports the development of distributed applications in heterogeneous network environments [5]. With SOC, applications are built by reusing existing third-party components or services that are invoked through specialized protocols. Precisely, the SOC paradigm heavily promotes component reuse in a loosely coupled way [1].

Ideally, with SOC, the client-side of service-oriented applications are made unaware of the details for invoking available services, such as interaction protocol, data-type formats, location, and so forth. In this sense, researchers are actively working towards providing tools and programming models to allow client applications to exploit services while enabling loose coupling. As Web Services [1] is the commonest technological choice for realizing the SOC paradigm, the terms "Web Service" and "service" will be used interchangeably throughout the rest of the paper.

SoC seems to be the right path towards providing truly loose coupling in service-oriented applications, as evidenced by the large body of Web Service invocation frameworks based on a diversity of traditional SoC techniques (e.g. source code annotations and aspect-oriented programming). These frameworks commonly succeed in hiding the physical details for invoking services, but they still fail at isolating client-side components from the "contracts", or interfaces, exposed by the services. This ineffective approach to SoC leads to in-house components that are subordinated to third-party contracts and must be modified and/or re-tested every time contracts change. In the end, SOC applications result much more difficult to modify and test. For example, a common requirement in SOC is to change the provider for a service, however *contract-subordinated* clients make this task significantly cumbersome.

In this paper we present a new approach to SOC development called DI4WS that builds upon SoC principles to establish looser relationships between client components and service contracts. Conceptually, we propose to treat third-party contracts as concerns and as such they should not be located within the boundaries of client applications. This concept is brought down-to-earth as a software layer placed between in-house components and services, in order to abstract the former from changes in the contracts. Accordingly, client applications can operate with different contracts by altering the intermediate layer, while the code implementing its components remains untouched. DI4WS combines the well-known Adapter and Dependency Injection (DI) [6] design patterns to provide developers with an intuitive programming model to non-intrusively switch among service providers in their applications.

The next section surveys previous SoC-based frameworks for developing service-oriented applications and explains how DI4WS improves over them. Section 3 overviews DI4WS. Section 4 discusses a case study, while Section 5 reports an experimental evaluation. Section 6 presents conclusions and future work.

2. RELATED WORK

From its beginnings, service-oriented development has to some extent relied on SoC. The WSIF [4] and the Apache CXF [1] follow a layered approach to isolate the code needed to invoke services, but they still are *contract-subordinated* approaches. Spring [2] hides concerns related to invoking third-party services, and allows developers to indicate how to map a particular contract onto client-side interfaces. To this end, developers supply the source code of their applications with annotations that tell Spring how to translate third-party operations to in-house methods and perform data-type mapping. The weak point of this approach is that contract details of specific providers remain bundled in such annotations, this is, the client code is still affected by changes in contracts.

In addition, there are academic efforts aimed at providing models and tools to further isolate service-oriented applications from services. WSSI [11] uses aspect-oriented techniques to dynamically replace a certain in-house method with a similar third-party operation. WSSI aims at fully automating the tasks of discovery and invocation of services at runtime, which has historically received criticism [12] as it is arguably difficult to incorporate an appropriate service into an application without any human intervention.

In this sense, some semi-automatic tools for service invocation have been proposed. WSML [2] employs a custom aspect-oriented language, named JAsCo, to introduce a software layer between applications and services. This layer deals with intercepting, adapting and forwarding client requests to services based on user-provided code in JAsCo. Though the authors have meticulously positioned their approach from a modeling perspective with respect to related research, the soundness of WSML has not been corroborated experimentally yet. Similar to WSML, the Daios framework [7] comprises a layered architecture that deals with many concerns related to service invocation beneath the application layer. Daios prioritizes efficiency and dynamism when invoking services, however its API, although simple, unavoidably leads to in-house components that depend on the structure of the messages implementing the operations of specific service providers. This is, Daios supplies a model for representing service inputs and outputs, which abstracts the target service's internals but not service's contracts.

Motahari Nezhad et al. [10] address this latter problem by semi-automatically generating Web Service representatives. To do this, [10] requires developers to specify expected contracts that are aligned with actual service contracts. Moreover, these representatives include framework placeholders so the programmer can manually customize data-type mapping and solve ambiguities, but this requires knowledge on the framework. Nagano et al. [9] refine this idea by making such specifications more generic and associating static stubs with them. By doing so, the same stub can bind to several Web Services, thereby enabling looser coupling between applications and services. However, service interfaces must be defined using a formalized XML and the generality required by the client-side specifications comes at the expense of requiring significant domain knowledge [7].

[1] `http://cxf.apache.org`

[2] `http://www.springsource.org`

Roughly, the above efforts can be grouped into two categories: those that aim at fully isolating client code from all aspects of Web Service invocation, including service contracts (i.e. WSML, WSSI, Nagano et al. and Motahari Nezhad et al.), and those that do not (i.e. the rest). The former group attempts to accommodate the interfaces of the services to be invoked to the ones specified and required by developers at design time. The latter group, while effectively pushes many aspects of service invocation out of the application logic through techniques such as layering (Daios, WSIF, Apache CXF) and annotations (Spring), promotes the idea of adapting the client code to the contracts of the invoked services. Consequently, the application is coupled to particular contracts and thus heavily depends on providers and their contracts, which in turn compromises the maintenance of client applications.

Note that these two groups represent in fact two different approaches for invoking Web Services, namely, contract-last and contract-first development of SOC client applications, respectively. The dichotomy, which has already installed an arduous debate when it comes to developing the server-side of SOC applications[3], has not been discussed in the context of client-side development yet.

As we mentioned earlier, existing efforts towards contract-last development, i.e. those belonging to the first group, are based on ad-hoc languages and programming models that are intuitively difficult to adopt. Unlike them, DI4WS merges the pervasive Adapter design pattern with Dependency Injection (DI), a popular programming style among developers [6]. Basically, the former serves as a mean to adapt the expected interface of potential services to the actual contract of a selected one, while the latter allows applications to be free from the client-side code that knows how to invoke third-party services. In SoC terms, the Adapter design pattern allows separating the code that depends on specific service contracts, while DI allows taking out the various administrative tasks involved in invoking services, from the client code.

3. DI4WS: FIRST ADAPT, THEN INJECT

An interesting implication of using DI in SOC is that client-side application logic can be isolated from the details for invoking services (e.g. URLs, namespaces, port names, protocols, etc.). 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 an external Web Service S with interface I_S from within an in-house component C, a dependency between C 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. At runtime, the code of C 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 to associate a configuration file with the client application, which is used by the DI container to determine which

[3] `http://www.infoq.com/articles/sosnoski-code-first`

components should be injected into other ones.

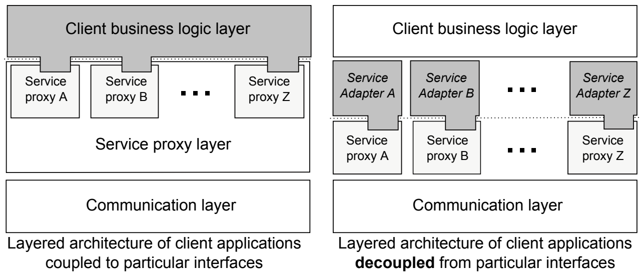
Client business logic layer
Service proxy A
Service proxy B
Service proxy Z
Service proxy layer
Communication layer
Layered architecture of client applications coupled to particular interfaces
Client business logic layer
Service Adapter A
Service Adapter B
Service Adapter Z
Service proxy A
Service proxy B
Service proxy Z
Communication layer
Layered architecture of client applications **decoupled** from particular interfaces

Figure 1: Architectural differences.

Although DI provides a fitting alternative to cleanly incorporate a Web Service into the source code of client applications, it leads to a form of coupling through which the application is tied to the invoked service contracts (i.e. the I_s interface). In this way, changing the provider for a service requires to adapt the client application to follow the new contract. This is illustrated from an architectural perspective in the left side of Figure 1, in which the layer containing the client business logic depends on particular proxies (grey layers are coupled to underlying ones). At the implementation level, this means to rewrite the portions of the application that use the replaced contract I_s, which includes operation signatures that are likely to differ from that of the new contract. This potentially implies different operation names and input/return data-types, which must be handled by the application by providing code explicitly.

To overcome this problem, DI4WS combines the DI and Adapter patterns 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), DI4WS injects a layer of *service adapters* (see the right side of Figure 1). A service adapter is a specialized Web Service proxy, designed through the Adapter pattern, which is responsible for adapting the contract of a particular service according to the contract (specified by the developer at design time) expected by the in-house components. We refer to A_{SC} as an adapter that accommodates the actual contract of a service S to the contract expected by an in-house component C. In other words, an adapter carries the necessary logic to transform the operation signatures of the expected client interface to the actual interface of a selected Web Service. For instance, if a Web Service operation returns a list of integers, but the application expects an array of floats, a service adapter would be responsible for performing the type conversion.

Service adapters support the notion of contract-last development of SOC client applications. As mentioned earlier, when following a contract-first approach to service consumption the application code is made compatible with the interfaces of the Web Services it uses. Under contract-last, service adapters accommodate the interfaces of the outsourced services to the interfaces designed by the developer but not yet implemented. In this way, changing a service does not affect the logic of the application, as it only requires to code another adapter for the new service.

The positive implications of DI4WS on maintainability can be quantitatively reflected through the Efferent Coupling (Ce) software metric [8]. Ce indicates how much the classes and interfaces within a package depend on classes and interfaces from other packages. In our case, this metric applies to all the components within the code of a client application depending upon external Web Services. When building loose-coupled SOC systems, the value of Ce should be as low as possible, since Ce represents the degree of dependency between the functional code of a client application and the interfaces representing server-side service contracts.

Formally, if a contract-subordinated SOC application with m internal components $\{C_1..C_m\}$ invokes n services $\{S_1..S_n\}$, then a value of $Ce_{i,j}$ equals to 1 means that the in-house component i is coupled to the contract of the j^{th} service. Then, the Ce value of contract-subordinated applications can be determined by $Ce = \sum_{i=1}^{m}\sum_{j=1}^{n} Ce_{i,j}$. When each service is invoked from exactly one component, Ce is equal to n. Instead, $n \leq Ce \leq n \times m$ when at least one service is invoked from two or more in-house components. Under DI4WS with the same assumptions, the Ce value is always equal to n. This is because the adapters are the only client-side components that depend on the service contracts and there is one adapter per service contract, i.e. n adapters. Accordingly, DI4WS reduces Ce in most cases, while not incrementing it in corner ones.

Besides reducing couplings, contract-last outsourcing allows developers to focus on the design, implementation and test of the in-house components, and then discover and incorporate the needed services. This separation may improve the development process itself, since these two groups of tasks can be performed independently by different development teams.

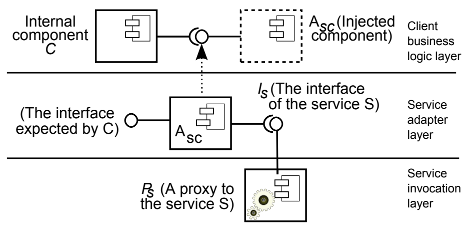
Internal component C
A_SC (Injected component)
Client business logic layer
I_S (The interface of the service S)
(The interface expected by C)
A_SC
Service adapter layer
P_S (A proxy to the service S)
Service invocation layer

Figure 2: DI4WS: Component model.

In contrast to other works that follow a contract-last approach to develop client applications, DI4WS does not require developers to learn neither new programming languages/frameworks nor models. Instead, DI4WS only requires to define at design time the interface(s) the in-house components will use to invoke third-party services. Then, at implementation time, developers code adapters for bridging the differences between the expected interfaces and those of the selected services. Graphically, this is shown in Figure 2, using the UML 2.0 notation for modeling components.

4. CASE STUDY

To understand the implications of modeling complex applications with DI4WS, next we will perform a qualitative comparison between two implementations of a real application. Unlike the experiments described in the next section, the purpose of this section is not to assess the efficiency of DI4WS for invoking services, but provide hints on how to design SOC applications with DI4WS, and perceive its implications in the resulting source code.

We separately followed contract-first and DI4WS approaches to develop a personal agenda that invokes Web Services. The personal agenda was in charge of managing a user's contact list, arranging new meetings, and to notify these contacts of new planned meetings. Below we list the tasks carried out by the personal agenda upon the creation of a new meeting. We assumed that the user of the personal agenda provides the date, time, participants and location of the meeting:

- *Getting a weather forecast* for the meeting place at the desired date and time.

- *Obtaining the driving directions* that each contact participating in the meeting could employ.
- For each participant of the meeting:
  - Including the weather report and the obtained route information in an email.
  - *Spell checking* the text of the email.
  - *Sending the email*.

The text in italics represents the functionalities that were delegated to third-party Web Services during the development of the two variants of the personal agenda. We employed real-world Web Services from the data-set described in [3]. Figures 3 and 4 depict the component diagrams of the contract-first and the DI4WS version of the example application, respectively. As contract-first does not isolate the design of the application components that invoke Web Services (in our case the PersonalAgenda component) from the interface of these services, the approach makes such components depend on the server-side contracts, this is, IWeatherByZip[4], IIMapQuestService[5], ISpellChecker[6] and IHtmlEmail[7].

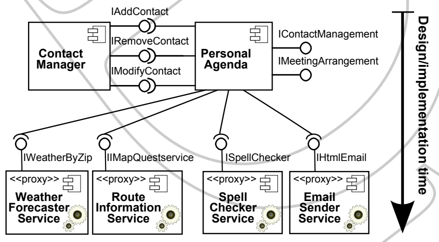

Figure 3: Component diagram of the contract-first agenda.

In opposition, when using DI4WS, the expected interfaces for potential services are specified at design time, whereas the bindings between these client-side interfaces and the server-side contracts are iteratively materialized at implementation time via our service adapters. Particularly, at design time the developer specifies the interfaces IForecast, IRouteInfo, ISpellChecking and IEmailSending.

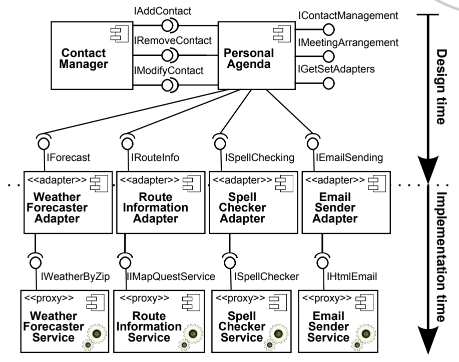
IAddContact
IRemoveContact
IModifyContact
Contact Manager
Personal Agenda
IContactManagement
IMeetingArrangement
IGetSetAdapters
Design time
IForecast
IRouteInfo
ISpellChecking
IEmailSending
<<adapter>> Weather Forecaster Adapter
<<adapter>> Route Information Adapter
<<adapter>> Spell Checker Adapter
<<adapter>> Email Sender Adapter
Implementation time
IWeatherByZip
IIMapQuestService
ISpellChecker
IHtmlEmail
<<proxy>> Weather Forecaster Service
<<proxy>> Route Information Service
<<proxy>> Spell Checker Service
<<proxy>> Email Sender Service

Figure 4: Component diagram of the DI4WS agenda.

All in all, the intermediate adapter layer of DI4WS allows for a better support in terms of maintainability when changing service providers. To exemplify this idea, let us suppose we want to use a different Web Service for sending emails, this is, another implementation for EmailSenderService (e.g. SendEmailService[8]). Consequently, IHtmlEmail is no longer valid, which impacts on the implementation of the contract-first variant. As suggested by Figure 3 this change forces to modify in the implementation of PersonalAgenda the code that invokes operations defined in IHtmlEmail and the object models of operation arguments. On the other hand, as DI4WS pushes the source code that depends on Web Service contracts out of the application logic via adapters, this change only requires to implement the corresponding parameter transformations within a new adapter and, in turn, inject it.

[4]`http://www.innergears.com/WebServices/WeatherByZip.asmx`
[5]`http://ws.strikeiron.com/MapQuestDrivingDirections2?WSDL`
[6]`http://ws.cdyne.com/spellchecker/check.asmx`
[7]`http://ws.acrosscommunications.com/Mail.asmx`

5. EVALUATION

This section describes experiments for measuring the impact of using DI4WS on performance and memory usage of applications. The motivation for this evaluation finds its roots in the fact that DI4WS introduces more indirections between applications and Web Services, which makes it potentially more inefficient than similar frameworks. To conduct the experiments we employed 4 applications, which were developed by last year software engineering students of a SOC course[9]. The applications invoked 6, 4, 4 and 7 services, respectively, which were deployed on a Tomcat container by using Apache CXF. The services had on average 3 operations with at least one using complex types or arrays. This enabled for a representative set of data-type transformations in the resulting adapters and thus provided the basis for a significant evaluation.

On the other hand, 3 alternatives to invoke the offered services from the applications were implemented: one using Daios [7], another one using Spring 2.5 and finally one using DI4WS. We chose these tools since Daios is a recent and well-published academic Web Service framework, while Spring 2.5 is a framework extensively used in the software industry. Moreover, the three alternatives promote rather different approaches for separating Web Service concerns from applications, namely layering, annotations, and object-oriented patterns, respectively.

We executed each application 20 times to measure the average execution time and memory consumption. To take time metrics, we computed the elapsed time by means of the "System.currentTimeMillis" primitive. The amount of sampling error of this primitive is 15 milliseconds, but it affects the time measurements associated with each application variant in the same way. To perform the allocated memory measurements, we used *jhat*, an utility included in Java 6.0 that can dump the entire object graph of a running application. The experiments were carried out on two computers connected through a 54 Mbps Wireless network with excellent link quality. We used a PC with an AMD Athlon XP 2200 (1.75 Ghz.) and 256 MB RAM to deploy the Web Services, and a laptop with an Intel Core2 T5600 (1.83 Ghz. per core) laptop with 1 GB RAM to run the client applications.

Table 1 shows the average amount of allocated memory of each variant per application. To clearly understand these results, it is worth mentioning some low level details about the runtime support of each variant. First, Spring 2.5 uses class introspection to dynamically proxy services, which results in extra objects and more allocated RAM. Similarly, Daios interprets the WSDL document of a service and loads a front-end to it into main memory. Contrarily, DI4WS directly instantiates concrete adapter classes for each service and injects them into the client application, thus reducing the size of the whole service representative. According to this, from

[8]`http://seekda.com/providers/abysal.com/SendEmailService`
[9]`http://www.exa.unicen.edu.ar/~cmateos/cos`

the table it can be seen that the variant using Spring 2.5 incurred in more allocated memory, followed by the variant using Daios. On the other hand, the variant using DI4WS required just a 61%-62% of the memory allocated by Spring 2.5.

	App. #1	App. #2	App. #3	App. #4
Daios	10.19	10.12	10.12	10.17
Spring 2.5	13.49	13.47	13.50	13.55
DI4WS	8.23	8.24	8.26	8.35

Table 1: Averaged allocated memory (MB) of each variant.

The DI4WS variants not only allocated less memory than its counterparts, but also run faster. Figure 5 shows the average execution time of each variant of the test applications. Again, for these applications, DI4WS outperformed Daios and Spring 2.5, though the adapters represent another software indirection between clients and services. These results may stem from the fact that Daios and Spring 2.5 variants dynamically build the client-side proxy needed to invoke a service the first time it is accessed, whereas DI4WS relies on static proxies, which are generated at implementation time. In other words, the execution time associated with either Daios or Spring 2.5 includes the time required to build proxies, because both frameworks generate proxies at execution time rather than development time. This is, they are designed for handling more dynamic invocation scenarios in which physical aspects of services such as location and protocol are determined at runtime, however service contracts are statically established in an code-invasive way.

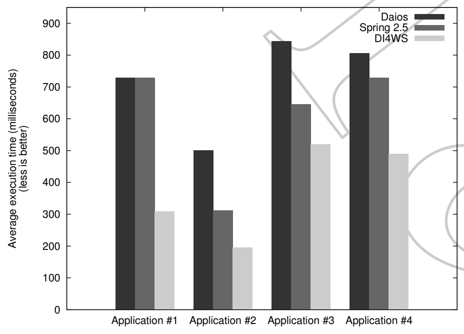

Figure 5: Averaged execution time of each variant.

6. CONCLUSIONS AND FUTURE WORK

The paper described DI4WS, an approach to help developers in building loosely-coupled service-oriented applications. Central to DI4WS is to treat the contracts of third-party service as a crosscutting concern. The proposed direction to deal with such a concern is to combine two widely-used design patterns. This is quite different to that of existing efforts, which are based on difficult-to-adopt SoC techniques, as it can be adopted by using any object-oriented programming language and DI container. Moreover, DI4WS nudges developers to follow a contract-last approach to invoke third-party services. Accordingly, the efferent coupling of SOC applications is reduced in most cases, while not incremented in corner ones, which has a positive effect on loose coupling and hence application maintenance. Besides, experimental results show that DI4WS does not compromise the performance of the resulting client applications.

This work represents a step towards answering two important research questions: a) how the steps of DI4WS for invoking services, namely adaptation and injection, impact on the software development process itself from an engineering point of view?, and b) can the combination of recent approaches to service discovery, which use the source code of client-side application components to build service queries [3], and DI4WS-based code help developers in discovering services more efficiently?.

With respect to a), we are planning to conduct experiments with several software development teams and larger applications, and various service modifiability scenarios, which will allow us to employ representative software metrics. Regarding b), the conjecture that drives this idea is that the client-side service interfaces obtained when using DI4WS may be general yet descriptive enough to retrieve potential Web Services. In fact, we have recently compared the recommendation efficiency for two service discovery systems and nearly 400 publicly available Web Services when using queries that were built from DI4WS-based applications and non-DI4WS ones. Preliminary results are encouraging and suggest that using DI4WS is also beneficial from a practical perspective, as it allows discoverers to quickly find proper services. Opportunities for future evaluations include using other service collections.

7. REFERENCES

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

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

[3] M. Crasso, A. Zunino, and M. Campo. Query by example for Web Services. In *Web Technologies track of the SAC'08*, pages 2376–2380. ACM, 2008.

[4] M. J. Duftler, N. K. Mukhi, and A. S. andSanjiva Weerawarana. Web Services Invocation Framework (WSIF). In *OOPSLA'01*. ACM, 2001.

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

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

[7] P. Leitner, F. Rosenberg, and S. Dustdar. Daios: Efficient dynamic Web Service invocation. *Internet Computing*, 13(3):72–80, 2009.

[8] R. C. Martin. Object-oriented design quality metrics: An analysis of dependencies. *Report on Object Analysis and Design*, 2(3), 1995.

[9] S. Nagano, T. Hasegawa, A. Ohsuga, and S. Honiden. Dynamic invocation model of Web Services using subsumption relations. In *ICWS'04*, pages 150–157, Washington, DC, USA, 2004. IEEE Computer Society.

[10] H. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW'07*, pages 993–1002. ACM, 2007.

[11] M. Pérez Reséndiz and J. O. Olmedo Aguirre. Dynamic invocation of Web Services by using AOP. In *ICEEE '05*, pages 48–51. IEEE Computer Society, 2005.

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

[13] C. Richardson. Untangling enterprise Java. *Queue*, 4(5):36–44, 2006.