Journal of Web Engineering, Vol. Vol., No. No. (Year) Page Nos.

EMPIRICALLY ASSESSING THE IMPACT OF DEPENDENCY INJECTION ON THE DEVELOPMENT OF WEB SERVICE APPLICATIONS

MARCO CRASSO, CRISTIAN MATEOS, ALEJANDRO ZUNINO and MARCELO CAMPO
ISISTAN Research Institute, Universidad Nacional del Centro. Also CONICET.
Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina.
Tel.: +54 (2293) 439682. Fax.: +54 (2293) 439681
mcrasso@gmail.com

Received (received date)
Revised (revised date)

Service-Oriented Computing (SOC) has been broadly conceived as the next big thing in distributed software development. The software industry has embraced SOC through Web Services –functionality that is accessible via ubiquitous protocols such as HTTP–. This technology provides the basis for reuse and interoperability of applications across the WWW. However, consuming Web Services is still an expensive task in terms of development costs, since developers still have to invest much effort not only into manually discovering services, but also on providing code to invoke them, which leads to software that is polluted with service-aware code and therefore is more difficult to modify and test. Recently, a technique that has become very popular for building software is Dependency Injection (DI), which allows applications to be far more testable and maintainable. In this paper, we quantitatively analyze some of the benefits and costs of DI for building Web Service applications. We base our experiments on a refined version of DI that combines text-mining, machine learning, and best practices from component-based software development to simplify the way Web Services are discovered and consumed. To our knowledge, this is the first study on the impacts of using DI in the context of SOC.

Keywords: service-oriented computing, Web Services, dependency injection, code-first outsourcing, text mining
Communicated by: to be filled by the Editorial

1 Introduction

Service-oriented computing (SOC) is a contemporary computing paradigm that supports the development of distributed applications in heterogeneous network environments [1]. SOC applications are built by composing existing functionalities (or *services*) that are published and accessed through specialized protocols. From the point of view of software engineering, SOC is an interesting paradigm for application development, since it heavily promotes software reuse in a loosely coupled way [1].

Mostly, the software industry has materialized SOC through Web Services [2], this is, programs with well-defined interfaces that can be located, published and invoked by means of ubiquitous Web protocols [3, 4]. The model underpinning Web Services encompasses three elements: service providers, service requesters and service registries. A service provider creates a Web Service description (commonly in WSDL [5]) and publishes it in a service registry (usually UDDI [6]). A service requester uses the registry to find a Web Service that matches

his needs, and then invokes its operations by using the corresponding WSDL description. Particularly, the aim of the WSDL and the UDDI standards is to provide interoperability among client applications and services across the Web.

Despite the important benefits Web Services provide, namely loose coupling between interacting clients and providers, and high levels of interoperability, they are still not as broadly used as one may expect [7, 8]. This is because current approaches to Web Service consumption require developers to *manually* look for suitable services and "glue" them in the client side code afterward. This practice forces developers not only to invest effort into performing discovery, but also to provide code to interact with selected Web Services, which leads to less maintainable software. Then, both the tasks of developing service-oriented software and carrying out its maintenance become harder.

On one hand, common materializations of UDDI supply developers with keyword-based search and category browsing of Web Services only. Due to the ever increasing number of publicly available services, finding proper services –i.e. those fulfilling the *functional* expectations of the client– through standard-based registries is like finding a needle in a haystack for a human discoverer [9]. This, in turn, has a negative impact on the cost of developing service-oriented software [10]. On the other hand, the common approach to call Web Services from within the application code is by interpreting its associated WSDL document with the help of invocation frameworks such as WSIF [11] or CXF [12]. Though these frameworks offer convenient programming abstractions for accessing Web Services, developers have to provide the necessary boilerplate code to invoke Web Service operations [13]. Consequently, applications result in a mix of pure logic and functionality for accessing Web Services.

A technique that has recently gained much popularity for building maintainable software is Dependency Injection (DI) [14]. With DI, a component can be incorporated (or *injected*) into another component without affecting the implementation of this latter. Empirically, it has been shown that the software using DI tend to have lower coupling than the software not employing DI [15], which has a positive impact on maintainability.

DI indeed allows developers to achieve high quality designs [14, 16]. Nevertheless, in the context of Web Service applications, DI is underexploited. An example of a DI-based framework for building SOC applications is Spring [14], which allows programmers to non-invasively bind their applications to RMI objects, CORBA applications and Web Services. Like Spring, Web Service frameworks follow a *contract-first* approach to service consumption: based on an input WSDL, i.e. the contract, they generate client side software artifacts (or stubs) for transparently proxying the remote service. Though this practice allows designers to separate business logic from the mechanisms for communicating with Web Services, the application code remains subordinated to particular Web Service contracts. Therefore, changing service providers requires to change the application logic as well, compromising modifiability and out-of-the-box testing and thus reducing the internal quality of the software. Besides, as the contracts of the Web Services to be used condition the way the application code is implemented, programming an application must unavoidably come after the necessary services have been chosen. In other words, these tasks cannot be carried out in parallel, thus potentially resulting in more cost to the whole development process.

We claim that DI can be further exploited to address the limitations of DI-based contract-first approaches to SOC development. We advocate for the use of DI along with a *code-first*

approach to service consumption: rather than having the application code subordinated to the contracts of the Web Services it accesses, we encourage developers to focus first on implementing and testing the functional code of their applications, and then to incorporate the necessary Web Services to "WS-enable" them. To this end, we combine DI and the Adapter design pattern [17] to establish loose relationships between clients and service providers, thus providers can be switched without affecting the application logic. The approach also takes advantage of the structure inherent to a DI application to efficiently look for required services. Recently, several works have proposed to assist human discoverers in finding proper services through standard-based registries by automatically improving the descriptiveness of discoverers' queries [10, 18, 19]. Unlike the approaches for semantically annotating source code and Web Services in an explicit manner [13], this alternative idea is to mine certain information that is implicitly conveyed in software artifacts [10, 18, 19]. Naturally, such "implicit semantics" cannot completely replace the need for explicit semantic descriptions in the context of systems developed via automatic Web Service discovery such as intelligent agents, in which applications consume services without human intervention [13]. Nevertheless, as the backbone of semantic approaches is describing each detail of available services and queries using machine interpretable languages, in practice this means gaining retrieval effectiveness at the expense of incrementing the cost for adopting SOC [7]. Contrarily, [10] has experimentally shown that the cost of Web Service discovery can be curbed by using the client side interface specification of a component that is to be outsourced when querying Web Service registries, but without the necessity of sacrificing either of the aspects involved in the "retrieval effectiveness versus cost of adoption" trade-off.

In this sense, we propose to use the information present in the relationships between the implemented DI-based application components to improve the descriptiveness of these queries and thus promptly retrieving proper third-party services. Intuitively, expanding queries based upon components with *strongly-related* and *highly-cohesive* operations should enhance the descriptiveness of service queries. We have materialized these ideas as an Eclipse plug-in that helps in "servifying" DI-based applications. This paper describes the underlying development model materialized by this tool, and provides an evaluation of the model to assess the benefits of simultaneously using DI and following a code-first approach to Web Service consumption for developing SOC applications. Basically, we quantified the impact of our model on Web Service discovery and the amount of system resources (i.e. execution time and memory) required to consume services under this model with respect to SOC tools not using DI and code-first service consumption. A major contribution of this paper is an assessment of the impact of using conventional DI and our refined DI for building Web Service applications. Essentially, the novel aspect of the paper is the provision of empirical evidence for software practitioners on the upsides and downsides of relying DI for building such kind of applications, a study that has not been explored until now.

The rest of the paper is organized as follows. The next section overviews DI. After that, Section 3 discusses the role of DI in the context of SOC applications. Then, Section 4 describes how DI can be exploited to ease the development of Web Service applications. Later, Section 5 presents a detailed evaluation of our model. Section 6 describes related works. Lastly, Section 7 concludes the paper.

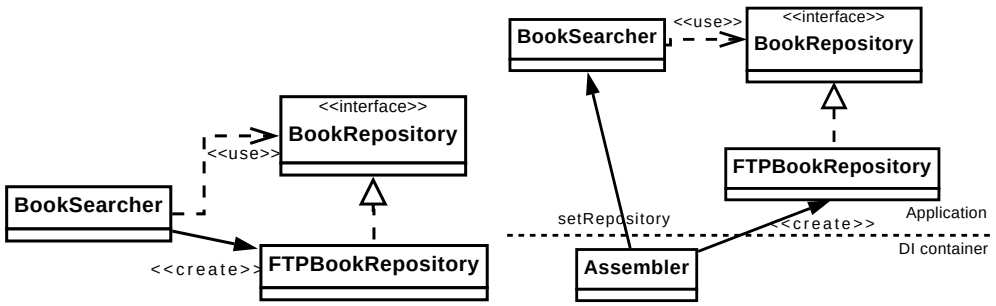
2 Dependency injection: An overview

Component-based development (CBD) is a branch of software engineering whose emphasis is on building applications in which functionality is split into a number of logical components with well-defined interfaces. Every component is designed to hide its associated implementation, to not share state, and to communicate with other components via message exchange. In the end, application components only know each other's interfaces, thus high levels of flexibility and reuse can be achieved [20].

A programming technique that has become very popular in CBD is Dependency Injection (DI) [14]. DI builds on the decoupling given by isolating components behind interfaces and focuses on delegating the responsibility for component creation and binding to a *DI container*. Basically, when a component C_1 accesses operations of another component C_2 (i.e. C_1 *depends* upon C_2), instead of explicitly creating and using an instance of C_2 within the code of C_1, the idea is to implement C_1 in such a way it can be externally bound to a concrete implementation of C_2. Precisely, a DI container is the run-time entity that injects dependant components like C_1 into provider components like C_2.

In the following paragraphs we will illustrate DI through an example. First, let us suppose we have an ordinary Java application for listing books of a particular author and topic (BookSearcher) that contacts a remote data repository where book information is stored as FTP-accessible files. Basically, the application opens a connection to the remote repository, transfers and parses the file(s), and then iterates the results locally to display book information. The source code of the components implementing our application is:

```
1  public class BookSearcher implements ...{
2    BookRepository repository = null;
3    public BookSearcher(String repositoryURL, String username, String password){
4      repository = new FTPBookRepository(repositoryURL);
5      (FTPBookRepository)repository.setUsername(username);
6      (FTPBookRepository)repository.setPassword(password);
7    }
8    public void displayBooks(String topic, String author){
9      List<Book> books = repository.getBooksByTopic(topic);
10     for (Enumeration<Book> elems=books.elements(); elems.hasMoreElements(); ){
11       Book book = elems.nextElement();
12       String fullName = book.getAuthorFirstName() + " " + book.getAuthorLastName();
13       if (fullName.equals(author))
14         System.out.println(book.getTitle());
15     }
16   }
17 }
18 public interface BookRepository{
19   public List<Book> getBooksByTopic(String topic);
20 }
21 public class FTPBookRepository implements BookRepository{
22   public String repositoryURL, username, password = null;
23   public FTPBookRepository(repositoryURL){
24     this.repositoryURL = repositoryURL;
25   }
26   // getters/setters for "username" and "password" go here
27   public List<Book> getBooksByTopic(String topic){
28     transferFile(repositoryURL + "/" + topic + "/" + "books-2008.dat");
29     transferFile(repositoryURL + "/" + topic + "/" + "books-2007.dat");
30     ...
31     returns parseAndJoinBookFiles();
32   }
33   protected void transferFile(String fileURL){
34     // FTP-specific instructions
35   }
36 }
```



(a) Without using DI (b) Using DI

Fig. 1. Class diagrams of the example application

Figure 1 (a) shows the class diagram corresponding to this code. Note that BookSearcher has to setup an instance of FTPBookRepository (lines 4-6) by providing it with some initialization parameters, namely the location of the repository (line 4) and authentication information (lines 5-6). Though BookSearcher is only interested in retrieving and then filtering books, it has to know implementation details of FTPBookRepository. As a consequence, BookSearcher is coupled both to the interface of the book repository (BookRepository, lines 18-20) and its concrete implementation (FTPBookRepository, lines 21-36).

Now, if we want to use a different mechanism for storing books such as a database or a Web Service wrapping the data, we have to modify the code of BookSearcher, and perhaps retest it. Depending on the way book information is accessed, a different set of configuration parameters could be required (e.g. database location, drivers, username and password for a database repository; URLs, ports and namespaces for a Web Service repository). Consequently, BookSearcher must also include the necessary constructors/setters methods. All in all, the cause of these problems is that the implementation of the BookSearcher component is not fully abstracted away from the mechanism (code and configuration) for accessing the repository. Expanding this into a real system, we might have dozens if not hundreds of such cases.

The DI version of the listing component is shown next. Essentially, BookSearcher now exposes a setRepository(BookRepository) method (lines 5-7) so that a DI container can inject the particular retrieval component being used[a] (line 3). Note that BookSearcher contains instructions only for browsing and filtering book information, and its source code neither depends on a concrete implementation of BookRepository nor includes protocol-specific configuration parameters. The DI version of BookSearcher is:

```
1  // The component into which an implementation of BookRepository is injected
2  public class BookSearcher{
3    BookRepository repository = null;
4    public BookSearcher(){}
5    public void setRepository(BookRepository repository){
6      this.repository = repository;
7    }
8    ...
9  }
```

[a]Most DI containers support two forms of injection: setter injection (components express dependencies by means of getters/setters) and constructor injection (components specify dependencies via constructor arguments). We will use setter injection throughout the rest of the paper.

Now, we must assemble the BookSearcher and FTPBookRepository components so that an operative application is built (see Fig. 1 (b)). Particularly, we have to indicate the DI container to use an instance of FTPBookRepository when injecting a value into the repository field of BookSearcher. This is supported in most containers by configuring a separate file (usually XML), which specifies a concrete implementation and configuration information for every component of an application and their dependencies. In the rest of the article, we will use Spring [14] as the DI container. Then, the configuration file for our application is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="mySearcher" class="BookSearcher">
    <property name="repository"><ref local="myRepository"/></property>
  </bean>
  <bean id="myRepository" class="FTPBookRepository"/>
    <constructor-arg value="ftp://nowhere.com:21/books"/>
    <property name="username">myUsername</property>
    <property name="password">myPassword</property>
  </bean>
</beans>
```

It can be seen from Fig. 1 (b) that after employing DI the dependencies to concrete classes within our application were removed. Also, the configuration parameters for these classes were moved from the code to a separate file, which is processed at runtime by an assembling element supplied by the DI container. In summary, we have obtained a better design in terms of coupling and cohesion.

3 The role of DI in SOC development

SOC concepts have evolved from component-based notions to face the challenges of software development in heterogeneous distributed environments [21]. A SOC application can be viewed as a component-based application that is created by assembling two types of components exposing a clear interface to their capabilities, as illustrated in Fig. 2[b]:

- *internal*: components locally embedded into the application,
- *external*: components that are either statically or dynamically bound to a service.

When building a new application, a designer may decide to provide an implementation for an application component, or to reuse an existing implementation (i.e. a service) instead. This latter practice is known as *outsourcing* [22].

In this context, to outsource a component C means to *fill the hole* left by the missing functionality with the one offered by an existing Web Service S. In the Web, there may be many published services that serve to this purpose. For example, if a component for providing current foreign exchange rates is needed, either ServiceObjects[c] or StrikeIron[d] services could be used. In this sense, an early problem is how to support effective and quick outsourcing of Web Services. Another problem is how to incorporate outsourced services into the internal components while achieving good quality attributes in the resulting software. DI offers opportunities for simultaneously tackling both of these problems.

[b] For modeling components, we will use from now on the UML 2.0 notation
[c] ServiceObjects `http://trial.serviceobjects.com/ce/CurrencyExchange.asmx?WSDL`
[d] StrikeIron `http://ws.strikeiron.com/ForeignExchangeRate?WSDL`

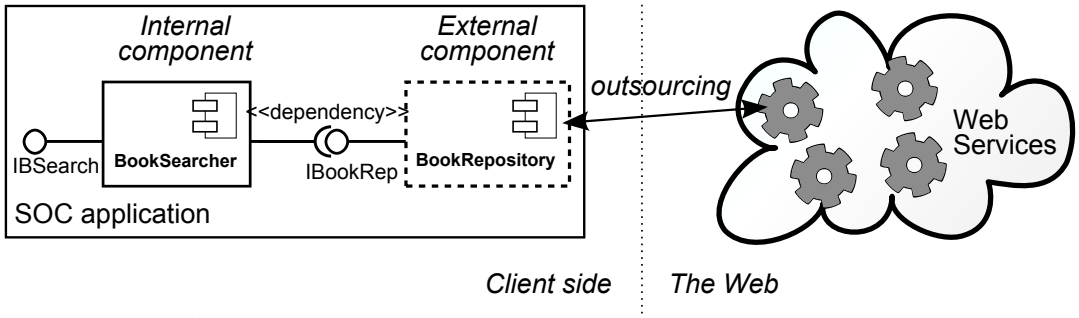


Fig. 2. Internal and external components in SOC applications

Typically, looking for services that fulfill a certain functionality in a registry, such as implementations of UDDI, is a time-consuming task when the number of services is large, which is precisely the case of massively distributed environments like the Web. As a consequence, the cost of developing SOC applications grows dramatically. However, a DI-enabled application contains a lot of information that can be exploited to speed up this process and curb its corresponding cost. For example, by analyzing the client side interface specification of the component that is to be outsourced (i.e. the signature of the operations expected by its related internal component(s)) the search space could be narrowed down so as to allow developers to select a service from a wieldy list of candidates [10]. Furthermore, results could be further refined by taking into account information about the context in which the component being outsourced is accessed, i.e. the internal component(s) from which those operations are invoked.

DI also provides a fitting alternative to *non-intrusively* incorporate a Web Service into the source code of internal, dependant application components. Concretely, when an internal component C_i accesses the functionality of an external component C_e, DI allows C_i to be unaware of the mechanisms to actually interact with the outsourced service associated with C_e. Then, any internal component can invoke Web Services just like if they were calling operations on another internal component, which makes service consumption more natural to the programmer and frees the application logic from having code tied to Web Service APIs or frameworks. In fact, the Remoting module of Spring provides a number of built-in components that can be injected into applications to easily exchange data with remote services. With this in mind, a developer thinks of a Web Service as any other regular component providing a clear interface to its operations, thus dependencies to external services by means of their interfaces can be established. In Spring, if a developer wants to call a Web Service S whose interface is I_s from within an internal component C, a dependency between C and S is established through I_s, causing a proxy to S that realizes I_s to be transparently injected into C.

4 DI4WS: Taking DI a step further

We observe that, to date, DI has not been fully exploited for building SOC applications. In this sense, we introduce DI4WS, a development model for building Web Service applications.

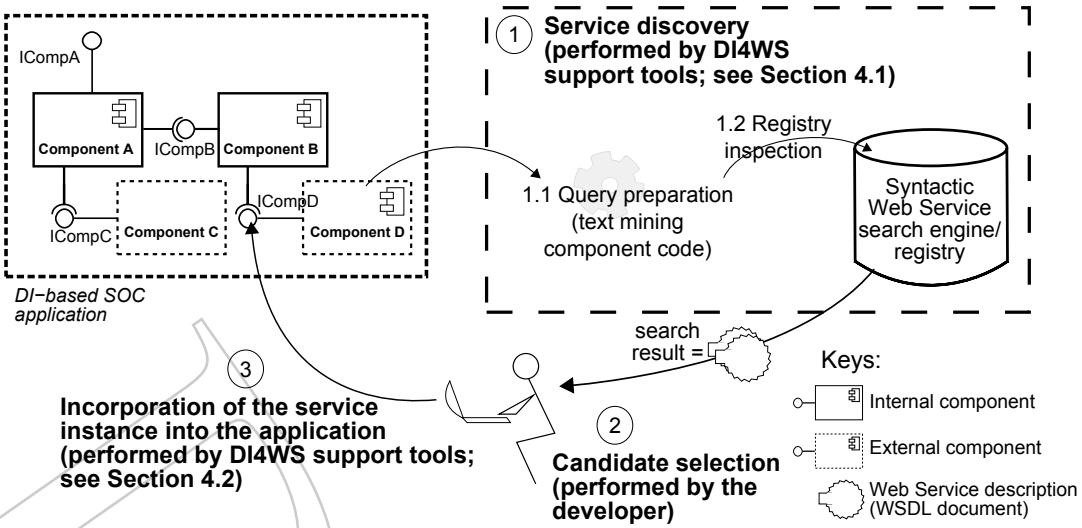


Fig. 3. Overview of DI4WS

An overview of DI4WS is shown in Fig. 3. DI4WS takes as input an incomplete DI application, where some of its constituent components are implemented, and others are intended to be outsourced to Web Services. In the figure, these two types of components are sketched with solid and dashed lines, respectively. Based on the dependencies between the (implemented) internal and the (non-implemented) external components of the input application, a semi-automatic process is iteratively applied to quickly and seamlessly associate an individual Web Service with each one of the external components.

Each iteration of this outsourcing process involves three steps: (1) finding the list of candidate services, (2) selecting an individual service from this list, and (3) injecting a representative of the selected service into the application to allow this latter to interact with the service at runtime. DI4WS provides developers with a GUI that performs steps (1) and (3) automatically and semi-automatically, respectively, whereas step (2) is in charge of users. Overall, the discovery-selection-injection sequence is performed until all external components of the input application have been associated with a Web Service.

As shown in Fig. 3, Step (1) consists of two sub-steps, namely sub-step (1.1) "Query preparation" and sub-step (1.2) "Registry inspection". The former deals with generating meaningful queries that describe those components that are intended to be outsourced to third-party services. Sub-step (1.2) in Fig. 3 represents the task by which a Web Service search engine is asked for services similar to the previously generated queries. This sub-step has been designed to be accomplished with any service discovery mechanism that takes as input a keyword-based query and returns a ranked list of WSDL documents, according to how similar to the query they are. The next section describes how this is done in the context of a syntactic Web Service search engine called WSQBE [10], which was also employed to perform the experiments. After the developer selects a service from this list (step (2)), DI4WS semi--automatically injects a special proxy to call the service at runtime. Step (1) is explained in

the next section. Step (3) is described in Section 4.2.

4.1 Step 1: Discovering Web Services

Syntactic approaches for Web Service discovery evolved from previous research on classic document retrieval. From an Information Retrieval (IR) viewpoint, the data within an information system include two major categories: documents and queries. The key problems are how to state a query and how to identify documents that match the query [23]. The cornerstone of syntactic approaches to document retrieval is to use a common representation for documents and queries, and to find out those with the most similar representations. Spatial representations allow mapping documents onto vectors in a Vector Space Model (VSM) [24], while preserving the locality of similar content, so that similar documents are mapped onto near vectors. Broadly, the VSM is an algebraic model for representing text documents in a multidimensional vector space, where each dimension corresponds to a separate term, usually single words. Therefore, by representing queries as vectors, discovering relevant documents operates by comparing such vectors.

With respect to Web Services, as both UDDI and WSDL standards are text-oriented, service descriptions can be seen as documents and then smoothly mapped onto vectors [10]. Keyword-based and natural language queries can be mapped onto spatial representations as well. In both cases, a collection of terms $d = \{term_0, term_1 ... term_n\}$ is mapped onto a vector $\vec{d} =< w_0, w_1 ... w_n >$, in which each component w_i indicates how representative of that document is an individual $term_i$. There are different term weighting schemes for determining the representativeness, or weight, of each distinct term [24]. A weighting scheme called Term Frequency (TF) determines that a term is important for a document if it occurs often in that document, i.e. the higher the frequency of the term within a document, the higher its importance [23]. TF-Inverse Document Frequency (IDF) is a combined scheme that rates as less important those terms that occur simultaneously in many documents [23]. In the end, WSDL documents and queries containing the same terms with similar weights will have similar spatial representations.

An important consideration when mapping a Web Service description onto the vector space is extracting terms from its associated WSDL document. The collection of terms gathered from a WSDL document comprises port-type names, operation names, in/out parameter names and comments [10, 18]. Subsequently, extracted terms are preprocessed to split them into combined words (e.g. splitting "getBooksByTopic" leaves "get", "Books", "By" and "Topic"), and then remove non-relevant words (also known as *stop-words*), bridge synonyms and remove the commoner morphological and inflectional endings from words (also known as *stemming*) [10]. Textual queries are frequently preprocessed in a similar way as well before matching them with available service representations. To illustrate how a VSM is built by a syntactic Web Service registry, such as [10], below we present an example. Let us suppose we have a service for calculating the factorial of any non-negative given integer, whose WSDL document is:

```
<message name="calculateFactorialRequest">
    <part name="originalNumber" type="xsd:long"/>
</message>
<message name="calculateFactorialResponse">
    <part name="factorial" type="xsd:long"/>
</message>
<portType name="FactorialCalculator">
```

```
    <operation name="calculateFactorial">
        <documentation>Returns the factorial calculator
              for a given number</documentation>
        <input message="tns:calculateFactorialRequest" name="number"/>
        <output message="tns:calculateFactorialResponse" name="result"/>
    </operation>
</portType>
```

For clarity reasons, the parts of the WSDL document that define concrete bindings to transport protocols, endpoints and namespace declarations have been omitted. By pulling out terms from the names and comments of the port-types, operations, messages and data-types, the registry gathers the following bag of terms: "calculateFactorialRequest", "originalNumber", "calculateFactorialResponse", "factorial", "FactorialCalculator", "calculateFactorial", "Returns", "the", "factorial", "calculator", "for", "a", "given", "number", "number" and "result".

Before building the vector space, the syntactic registry processes the collection of terms to "clean" them up. First, the registry divides combined words into single terms, e.g. splitting "calculateFactorialRequest" leaves "calculate", "factorial" and "request". Second, the registry removes articles, numbers, symbols, sentence connectors and other words with a low level of usefulness within the context of Web Services, e.g. the word "request". Finally, the registry applies Porter's stemming algorithm [25] to each single term. Now, the registry uses the resulting stems to represent the associated WSDL document in the VSM according to the TF weighting scheme. The resulting vector is:

$$\vec{v}_0 \quad = \quad (< factori, 4 >, < calcul, 3 >, < origin, 1 >)$$

Now, let us suppose that a user wants to find services relevant to the query: "calculate factorial". Then, the registry preprocesses the query and associates to it the following vector:

$$\vec{q} \quad = \quad (< factori, 1 >, < calcul, 1 >)$$

Subsequently, the registry finds vectors nearly located to $\vec{q}$ by using a distance measure such as the cosine similarity [23].

Central to discovery in DI4WS is the idea of mining source code files to extract terms that may enhance the descriptiveness of a query, thus potentially increasing the efficiency of Web Service discovery. In fact, this source code is DI-enabled code. Specifically, DI4WS employs a text mining process comprising five activities for mining relevant terms from the DI-enabled source code that implements an application, as we explain below.

As an illustration, let us suppose that the book searcher application described in Section 2 now requires to outsource a Web Service for obtaining information about books covering a topic, instead of using an FTP repository. Figure 4 depicts the detailed design of the application. Here, BookSearcher implements the internal component and uses the BookRepository interface, which represents the required external service. Book information is modeled through the Book class. The figure also shows three tables, in which each cell contains the output generated by each individual text mining activity after processing each source code file.

As the reader can note, we followed good naming and documentation practices for implementing the application. In particular, each class of the figure has been supplied with documentation in the form of a short natural language description of its intended purpose. Furthermore, we used self-explanatory names for classes, methods and properties. As shown in the columns of the tables, the three initial activities of our text mining process enlarge the

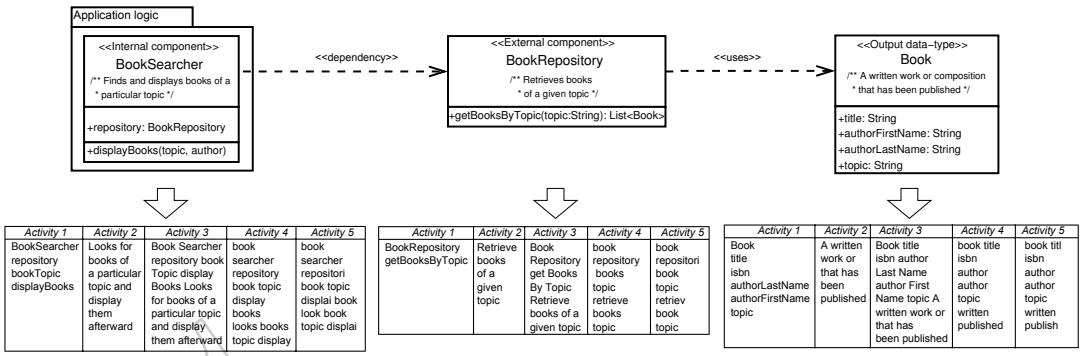


Fig. 4. Mining relevant terms from the book searcher example

collection of extracted terms, and as the process advances, the resulting collection of terms is refined. The union of the last column of these three tables suggests that the subsequent query expansion activities quantitatively improved the resulting query by adding terms (*stems* to be precise) belonging mostly to the book domain.

Following good practices when building component-based software results in components with strongly-related and highly-cohesive operations [26]. Based on this, we assume that the logic of a well-designed component commonly belongs to a unique domain, which is the same domain of those components that directly interact with it. For example, the component for providing information about books might be useful for components belonging to the library domain, while it rarely might be useful for a component in the business domain. Clearly, the source code files of these components may convey relevant information for finding services. Therefore, DI4WS expands queries by mining relevant terms from the source code files of both the non-implemented components and the internal components that directly depend on them.

Documenting source code and using descriptive naming conventions are also considered good development practices [27]. DI4WS assumes that, throughout their projects, developers use self-explanatory names for component properties, operations and arguments, and avoid using meaningless names like "arg1" or "foo". This assumption often holds, because from a developers' point of view the choice of meaningful identifiers is as important as documenting code [28]. However, by blindly mining a source code file, many irrelevant terms may be extracted (e.g. reserved words such as "if", "while", "return", etc.), thus degrading the descriptiveness of a query. To overcome this problem, DI4WS cleans mined terms as explained next.

Under the assumption of good naming and documenting practices, the interfaces describing external components have meaningful terms. We have designed a text mining process for pulling out relevant terms from the client side source code. Although our techniques may be used with any programming language provided developers follow these good practices, we have at present materialized it by using Java due to its high popularity. Generally speaking, the process takes any Java class or interface as input and generates a collection of extracted stems.

In a first activity, we pull out the name of a component and the name of its operations. Simultaneously, we mine developers' comments from Javadoc[e] elements. At this stage of the text mining process we have a collection of words and combined words, because commonly used notation conventions suggest to merge words by means of uppercase characters, numbers or hyphens (e.g. *getBooksByTopic*, *author_name*). Then, in a third activity we split all combined words. In a fourth activity, we remove symbols and stop-words. Finally, we utilize Porter's stemming algorithm [25] for removing the commoner morphological and inflectional endings from words, reducing English words to their stems. As a result, the output of this text mining process is a set of stems extracted from the specification of a class. Graphically, this process is shown on the right side of Fig. 5.

Under the assumption of strongly-related and highly-cohesive operations, the classes describing internal components (i.e. dependants) may contain terms related to the domain of the external service. Likewise, the classes describing arguments (inputs and outputs indistinctly) of any operation offered by the potential service, i.e. the external component, may also convey relevant terms. For example, in the book searcher example, the Book class stands for an output argument of the external component and contains 7 relevant terms.

To expand queries with relevant terms from argument classes and dependant components, we must first identify them. Once target classes have been found, we feed them to the text mining process. We have developed a plug-in for the Eclipse SDK that provides a graphical tool to simplify as much as possible the outsourcing process. The left side of Fig. 5 shows some screen-shots of the plug-in. A wizard opens when a user selects "Find services for..." by clicking on a Java interface from its source code. The wizard uses the Eclipse JDT [29] Search Engine for automatically selecting the components that depend on the interface and presenting them to the user. Then, the developer may select or discard some of these classes. Similarly, the wizard presents a list of argument classes to the user. This list is automatically built by analyzing the external component being outsourced and retrieving the class names associated with each argument of its operation(s), without taking into account neither primitive types nor classes provided by Java. Once the user has selected target classes, the wizard uses them along with the aforementioned Java interface as input for the text-mining process. Finally, the wizard sends the generated query to a service registry and presents the results as a candidate list.

Regardless of what sources of relevant terms we use or how we combine them, the output of the text mining process is always a collection of stems. Intuitively, the more the preprocessed components, the more the resulting stems. In other words, by combining several term sources we may augment the collection of the stems that constitutes a query. It is worth noting that under a non-DI or contract-first approach the classes for modeling the external component and its arguments are commonly absent until a candidate service is selected. On the other hand, DI-based applications consists of components that are isolated behind public interfaces that, in the context of SOC, represent external services. Then, a DI-based application contains more source code files for feeding the aforementioned text mining process, whereas an ordinary application contains only classes representing dependant components. In Section 5 we will evaluate how the inclusion of this extra information –namely, the interfaces and their argument types– impacts on the accuracy of service discovery.

[e] Javadoc is a tool for generating API documentation from comments in source code.

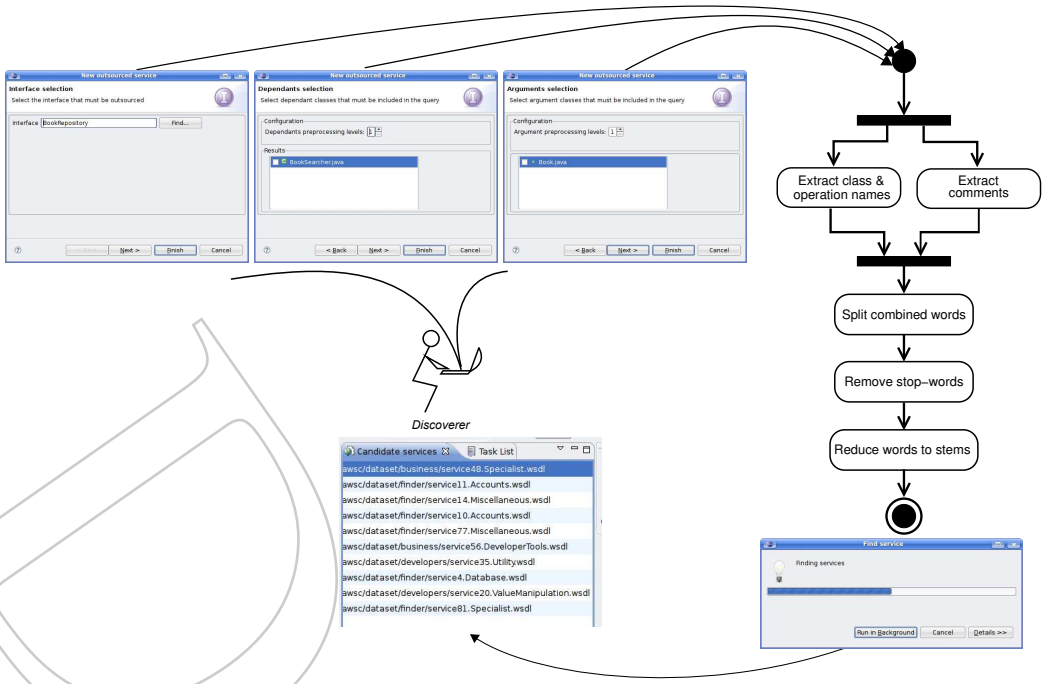


Fig. 5. The DI4WS Eclipse plug-in: Query generation and service selection

The next section focuses on describing in detail how discovered services, once selected, are consumed by applications under DI4WS.

4.2 Step 3: Incorporating a candidate Web Service

Contract-first Web Service outsourcing is based on the idea of adapting the client side code to the interfaces exposed by the services an application accesses (i.e. the contracted ones). Put differently, when an internal component C needs the functionality of a Web Service with interface I_s, the code of C will end up calling any of the methods declared in I_s. As explained before, this approach to service consumption can be employed in conjunction with DI, being Spring an example of a framework simultaneously providing both capabilities.

Unfortunately, contract-first outsourcing leads to a form of coupling through which an application is tied to the contracts (e.g. I_s) of the particular Web Services it relies on. In consequence, changing the provider for a service requires to adapt the application to follow the new Web Service contract. At the implementation level, this means to rewrite the portions of the application code that use the original service interface, which include operation signatures that are likely to differ from that of the new interface. A different operation signature may imply different operation names, or input and return data-types, which must be handled by providing code explicitly.

To overcome this problem and still leverage the benefits of DI, DI4WS refines the idea of Web Service injection by introducing an intermediate layer that allows developers to seamlessly use different services in their applications. Roughly, instead of directly injecting raw

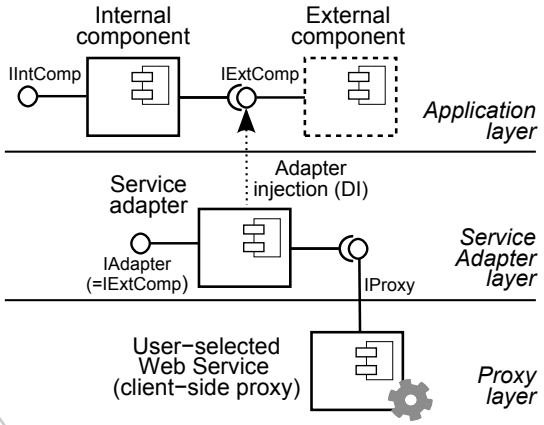


Fig. 6. Code-first, DI-based service outsourcing in DI4WS: Service adapters

Web Service proxies into the application, which demands to write the code of internal components in such a way it is compatible with the service contracts, DI4WS injects *service adapters* (see Fig. 6). A service adapter is a specialized Web Service proxy, designed through the Adapter pattern, which is precisely responsible for adapting the interface of an underlying service according to the interface (specified by the developer at design time) of the corresponding external component. Service adapters carry the necessary logic to transform the method signatures of the client side interface to the actual interface of the corresponding Web Service selected by the developer at step (2) of the DI4WS outsourcing process. For instance, if a Web Service operation returns a list of integers, but the application expects a float array, a service adapter would be responsible for performing the type conversion.

Service adapters support the notion of code-first service outsourcing. As mentioned previously, under contract-first the application code is made compatible with the interfaces of the Web Services it uses. In opposition, under code-first service adapters accommodate the interfaces of the outsourced services to the interfaces expected by the application, that is, the ones designed by the developer. In this way, changing a service does not affect the logic of the application, as it only requires to code another service adapter for the new service. Besides reducing couplings, code-first outsourcing allows developers to design, implement and test the code of the internal components, and then focus on the outsourcing of the external components. This separation contributes to improve the development process itself, since these two groups of tasks can be performed independently by different development teams.

To illustrate these ideas, let us return to the example of the book searcher application discussed in Section 2. The application was composed of two components: an internal component (BookSearcher) and an external component whose expected operations were specified by a BookRepository interface. After the developer chooses a Web Service for this latter, DI4WS generates a client side proxy to the service, the corresponding service adapter, and the configuration that is needed to inject the proxy and the service adapter into the application. The resulting classes for the application are shown in Fig. 7. In general terms, a service adapter can be associated with many internal components, because more than one internal

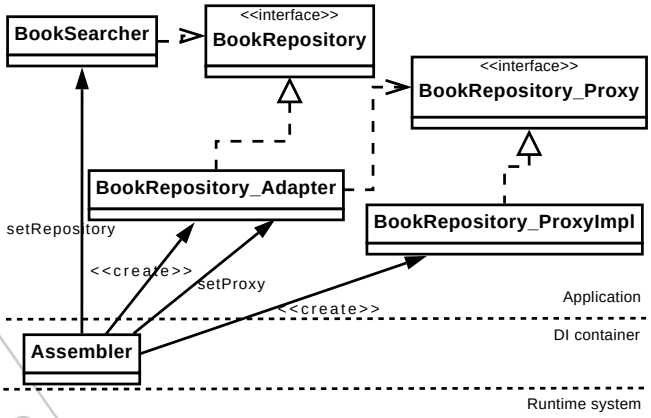


Fig. 7. The DI4WS book searcher application: class diagram

component may depend on the same external component.

The proxy to a selected Web Service is created based on the retrieved WSDL description of the service. Consequently, the proxy holds the logic to talk to the service. The interface of the proxy is exactly the same as the service contract established by the provider, which, under a code-first approach to outsourcing, will not usually be truly compliant to the service contract expected by the application (i.e. BookRepository). Currently, our plug-in automatically generates proxies via the Web Tools Platform (WTP) [30], which extends Eclipse with facilities for developing Web and Java EE applications. Each service adapter is partially generated by the plug-in as a class skeleton that bridges the interface of the client side proxy to the service (i.e. the service contract) to the interface expected by the application. In our example, the service adapter would automatically realize BookRepository, as it is injected into BookSearcher. The code to map any call to methods from this skeleton class to the proxy must be implemented by the developer. Although this task is carried out manually, the plug-in provides them with an editor for depicting which parts of the contract must be adapted (Fig. 8) . For instance, let us assume that the interface of the generated client side proxy is:

```
public interface BookRepository_Proxy{
  public BookInfo[] doKeywordSearch(String topic);
}
```

Here, doKeywordSeach is an operation derived from the WSDL description of the Web Service. In this way, the service adapter must map individual calls to the getBooks operation (application-side contract) to a call to doKeywordSearch (server-side contract) on the proxy, thus the code of the service adapter would be:

```
1  public class BookRepository_Adapter implements BookRepository{
2    private BookRepository_Proxy proxy = null;
3    public void setProxy(BookRepository_Proxy proxy){ this.proxy = proxy; }
4    public BookRepository_Proxy getProxy(){ return proxy; }
5    public List<Book> getBooksByTopic(String topic){
6      // User-supplied code starts here
7      Vector<Book> adapterRes = new Vector<Book>();
8      BookInfo[] adapteeRes = getProxy().doKeywordSearch(topic);
9      for (int i=0; i<adapteeRes.length; i++){
10         adapterRes.addElement(createBookInstanceFrom(adapteeRes[i]));
11     }
```

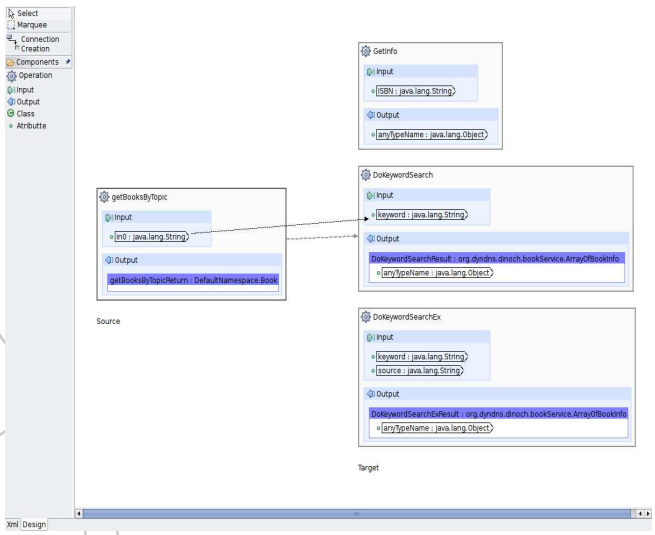


Fig. 8. The DI4WS Eclipse plug-in: Contract adaptation

```
12        return adapterRes;
13        // User-supplied code ends here
14    }
15  }
```

The service adapter implemented by BookRepository_Adapter only performs the translation of the invoked operation name and its return data-type (lines 8-11). However, the mapping task may also involve converting the input arguments of one or more adapter operations to the parameters of the corresponding proxy operations. Besides mapping data-types, service adapters are useful to include extra parameters when calling the operations of a service that otherwise would be contained in the application code, such as username/password or licensing information. Another advantage of using service adapters is that the code related to handle exceptions when invoking services remains isolated from the application logic.

Finally, our plug-in generates the DI-related configuration to wire the Web Service proxy, the service adapter and the internal component(s) using the service together, so as to build a complete application. A dialog lets developers to select one or more dependant components in which the external service will be injected (Fig. 9, left). For instance, the service providing book details would be only injected into the BookSearcher component. Optionally, a second dialog (Fig. 9, right) allows developers to set the class name for the service adapter and to establish DI-container settings (e.g. the location of the Spring XML configuration file). The resulting XML configuration for the book listing application is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="mySearcher" class="BookSearcher">
    <property name="repository"><ref local="repository_Adapter"/></property>
  </bean>
  <bean id="repository_Adapter" class="BookRepository_Adapter">
    <property name="proxy"><ref local="repository_Adapter_Proxy"/></property>
```

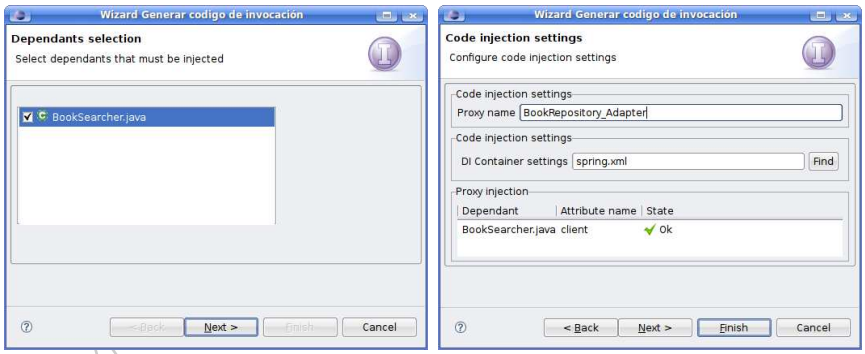


Fig. 9. The DI4WS Eclipse plug-in: Code generation and service adapter injection

```
  </bean>
  <bean id="repository_Adapter_Proxy" class="BookRepository_ProxyImpl"/>
</beans>
```

The configuration instructs the DI container to inject an instance of the generated service adapter into the corresponding dependant component (BookSearcher), and also an instance of the Web Service proxy (BookRepository_ProxyImpl) into the service adapter.

4.3 A case study: A personal agenda

In the next paragraphs we illustrate DI4WS for modeling more complex applications than the one explained up to now. Basically, we perform a qualitative comparison between the implementation of a service-oriented application based on contract-first service outsourcing, in which coding the application logic comes *after* knowing the contracts of the external services to be consumed, and DI4WS, which promotes code-first service outsourcing. Unlike the experiments described in the next section, the purpose of this section is not to assess the effectiveness and efficiency of DI4WS for outsourcing Web Services, but provide some 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 invoked 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. The contact list was 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 implementing the two variants of the personal agenda, we simplified the logic 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 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. Algorithmically, arranging a new meeting roughly involves:

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

- For each participant of the meeting:
  - Creating an email with an appropriate subject, and a body including the weather report and the obtained route information.
  - *Spell checking* the text of the email.
  - *Sending the email*.

The text in italics represents the functionalities that were outsourced to Web Services during the development of the two variants of the application. To bind these functionalities to services, we employed real-world Web Services from the data-set described in Section 5, which was also used for evaluating DI4WS. Figures 10 and 11 depict the component diagrams of the contract-first and the DI4WS version of the application, respectively. As contract-first does not isolate the design of the application components that directly consume Web Services (in our case the PersonalAgenda component) from the interface of these services, the approach makes such components to depend on the server-side contracts, this is, IWeatherByZip[f], IIMapQuestService[g], ISpellChecker[h] and IHtmlEmail[i].

In opposition, when using DI4WS, the expected interfaces for the outsourced Web 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 with the assistance of our plug-in. Particularly, at design time the developer specifies the interfaces of ContactManager and PersonalAgenda (internal application components) and the expected interfaces of the required Web Services or external application components, this is, IForecast, IRouteInfo, ISpellChecking and IEmailSending. In addition, the components that directly interact with Web Services include the corresponding interface to support DI, which have been modeled in Figure 11 through the IGetSetAdapters interface for PersonalAgenda. This interface is then accessed by the framework-level components that actually perform DI to link PersonalAgenda with instances of each one of the four adapters. At implementation time, the developer must include the getter and setter methods specified by the IGetSetAdapters interface.

All in all, the intermediate adapter components included by DI4WS allow 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, to have another implementation for EmailSenderService (e.g. SendEmailService[j]). As a consequence, IHtmlEmail is no longer valid, which impacts on the implementation of the applications. Particularly, in the contract-first variant, this change forces to modify in the implementation of PersonalAgenda the specific code that invokes operations defined in IHtmlEmail and the object models of operation arguments. On the other hand, as DI4WS effectively pushes the source code that depends on Web Service contracts out of the application logic via adapters, changing the email service only requires to rebuild its associated adapter and to implement the corresponding parameter transformations within the adapter. This, in turn, has a positive effect on the maintainability and testability of the internal components of the application.

[f] WeatherByZip `http://www.innergears.com/WebServices/WeatherByZip.asmx`
[g] IMapQuestService `http://www.xmethods.net/ve2/ViewListing.po?serviceid=98418`
[h] SpellChecker `http://ws.cdyne.com/spellchecker/check.asmx`
[i] HtmlEmail `http://ws.acrosscommunications.com/Mail.asmx`
[j] SendEmailService `http://seekda.com/providers/abysal.com/SendEmailService`

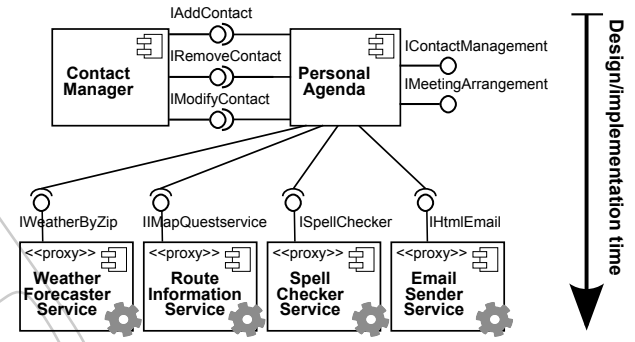


Fig. 10. Component diagram of the contract-first implementation of the personal agenda

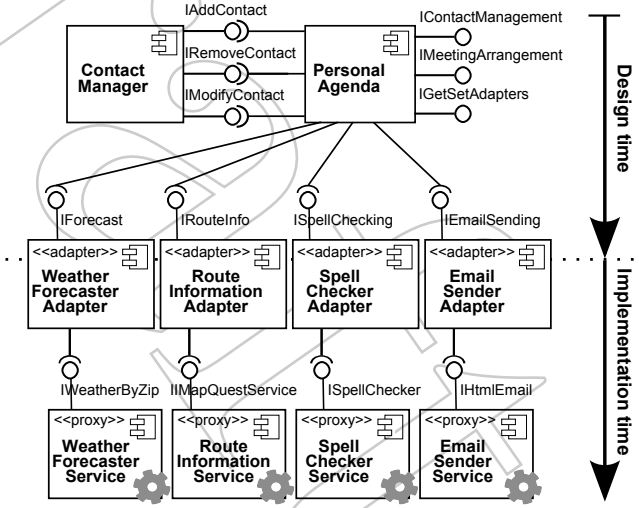


Fig. 11. Component diagram of the DI4WS implementation of the personal agenda

5 Evaluation

This section presents experiments that were carried out to provide empirical evidence about the impact of DI4WS. We focused on evaluating the effects of employing DI on: (1) Web Service discovery and (2) runtime performance. Concretely, we compared the effectiveness of discovery with two variants of a syntactic approach to Web Service discovery when generating queries from applications employing DI versus applications not employing DI, according to classic IR metrics. In addition, we assessed the overhead associated with supporting text mining and service adapters in terms of execution time and memory consumption, respectively. For a detailed report on the performance of the employed service registries, see [10].

To conduct the tests we built several service-oriented applications by outsourcing various Web Services, from which 30 queries were derived to be used as the evaluation-set. Each query was associated with at least two source code files. One source code file of each query consisted of the Java interface standing for an *injectable* external component, i.e. a dependency. Both the header and the methods of each dependency included proper comments. For those dependency methods that accepted user-defined data-types as arguments, their associated class files were properly commented as well. The other source code file represented the internal component that depended on the injectable one. Likewise, both the header and the methods of each dependant included comments. These sets of code files allowed us to evaluate two different types of term sources for querying the service registry, namely those resulting from mining DI-based source code, and those extracted when mining non-DI code.

For the service registry, we used two variants of the syntactic service discovery approach presented in [10] by using TF and TF-IDF (see Section 4.1), two well-known term weighting schemes [23], which are extensively used by syntactic approaches for service discovery [9, 10]. Then, we feed each variant of the registry with a publicly available collection of 391 categorized real-world Web Service descriptions in WSDL. These WSDL documents were gathered from repositories on the Internet by Hess et al. [31]. Once the same data-set was published in the two registry variants, DI-based and non-DI queries were used to retrieve services from them. Note that using the same collection of third-party WSDL documents for querying the registry with either DI-based code or non-DI code means that each experiment is evenly influenced by the peculiarities of the employed collection. For example, if the developer who published a service used meaningless and poorly descriptive names for naming parameters and operations within the corresponding WSDL document, e.g. "arg0" or "foo", then the retrieval effectiveness of both experiments –i.e. with DI-based and non-DI queries– will be harmed in the same way. Then, the utilization of the same real-world WSDL documents enables for a more fair comparison in terms of retrieval effectiveness.

We used Recall-at-n, Normalized Recall (NR), R-precision and Precision-at-n measures to evaluate the proportion of relevant services in the retrieved list and their positions relative to non-relevant ones [23] (see Table 1 for a brief summary of them). As some of these measures require to know exactly the set of all services in the collection relevant to a given query, we have exhaustively analyzed the evaluation-set to determine the relevant services for each query. To do this, we judged whether the operations of a retrieved WSDL document fulfilled the expectations previously specified in the Java code or not. For example, if a dependant required a component for converting from Euros to Dollars, then a retrieved component for converting from Francs to Euros was non-relevant, even though these components belonged

Table 1. Information Retrieval measures

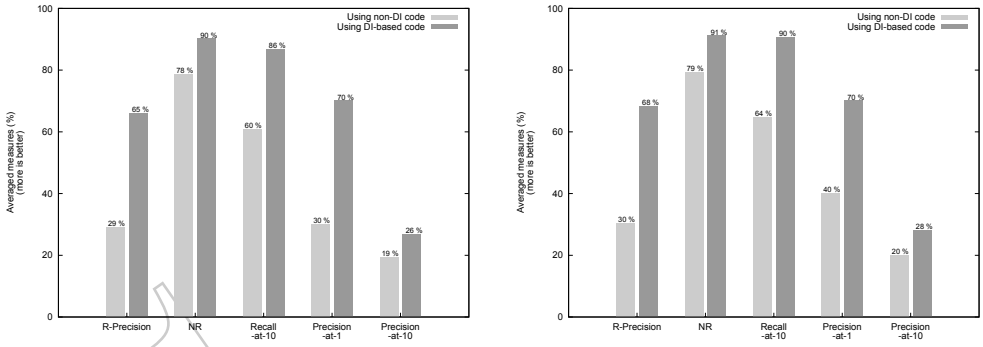
Measure	Description	Formula
Recall-at-n	Computes the proportion of retrieved relevant documents ($RetRel$) within a result list of size=n, where R represents all relevant documents in the evaluation-set	$\frac{RetRel_n}{R}$
Normalized Recall (NR)	Regards Recall and the position (r_i) of each relevant document (i^{th} relevant document) in the result list, where N is the size of the evaluation-set	$1 - \frac{\sum_{i=1}^{R} r_i - \sum_{i=1}^{R} i}{R(N-R)}$
R-precision	Computes the precision at the R position in the ranking	$\frac{RetRel_R}{R}$
Precision-at-n *(with n=1,10)*	Computes precision at the first and tenth positions of the result list	$\frac{RetRel_n}{n}$

to the same category or they were strongly related. In this particular case, only components for converting from Euros to Dollars were considered relevant. Note that this definition of hit makes the validation of our discovery mechanism very strict. Furthermore, it is worth noting that for any query there were, at most, 8 relevant services within the evaluation-set. Besides, there are 10 queries that had associated only one relevant service.

We computed the aforementioned measures for the 30 queries according to four scenarios, this is, using DI-based code and non-DI code in conjunction with either variants of the registry. First, we averaged the results when generating queries from the source code files related to dependencies, arguments and dependants, i.e. the set of files associated with the DI-based code. Second, we tested using *only* the dependant source code files as inputs of the query generation process, i.e. the files associated with the non-DI code, and averaged the results for the same measures. In a first round of tests, we omitted the natural language comments present in the corresponding codes for the four previous scenarios, this is, the tests were carried out with undocumented DI-based code and undocumented non-DI code. The reason for performing these tests was to evaluate whether the DI-based code may be more benefited from good practices on commenting source code than its non-DI counterpart, since DI-derived queries are generated by processing more Java source files, which in turn means including more comments. Therefore, for the sake of fairness, we decided to compare the implications of using DI versus not using it, when the code is undocumented.

Figure 12 shows the averaged results for each computed measure in groups of two bars for each service variant. The left/right bar within each group shows the achieved values for each considered measure when using non-DI/DI-based queries. Figure 12 shows that, for this evaluation-set, all measures achieved better results (higher values are better) when using DI-based code with both the TF and TF-IDF variants of the service registry, even though the comments in the source files were ignored when deriving the queries.

Table 2 presents a summary of the achieved results. Recall-at-10 results point out that, with DI-based code, one relevant service was ranked in a window of 10 retrieved services on the 86.63%-90.56% of the cases, according to each variant of the registry. When using non-DI code, on the other hand, these percentages fell down to 60.83%-64.63%. Similarly, Precision-at-1 results show that a relevant service was ranked first on the 70% of the cases when using DI-based code regardless of the variant of the registry. Instead, when using non-DI code, a relevant service was at the top of the result list on the 30%-40% of the cases. NR



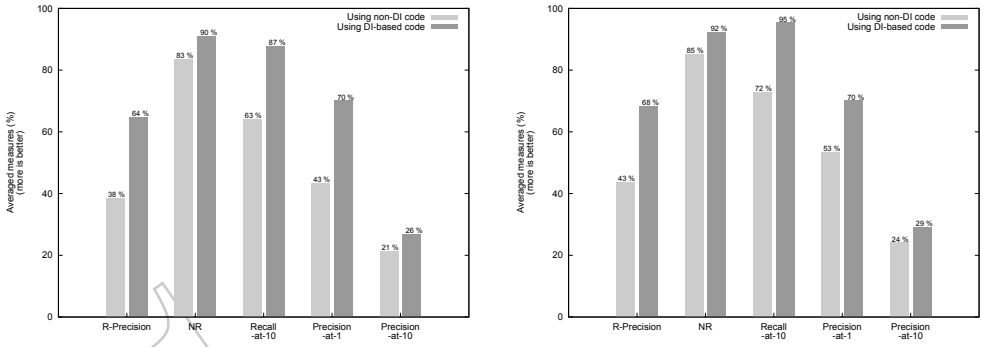
(a) Using the first variant of the registry (TF) (b) Using the second variant of the registry (TF-IDF)

Fig. 12. Average achieved results (%) for Recall-at-10, NR, R-Precision, Precision-at-1 and Precision-at-10 measures, when using uncommented source code

Table 2: Summary of achieved results

	DI-based		Non-DI	
Measure/Registry	TF	TF-IDF	TF	TF-IDF
R-Precision	65.82%	68.24%	29.07	30.38%
NR	90.07%	91.08%	78.54%	79.23%
Recall-at-10	86.63%	90.56%	60.83%	64.63%
Precision-at-1	70.00%	70.00%	30.00%	40.00%
Precision-at-10	26.67%	28.00%	19.33%	20.00%

and R-Precision show that both DI-based alternatives surpassed their non-DI counterparts, which means that more relevant services are ranked before non-relevant ones when using DI. Precision-at-10 results also present improvements when using DI-based code with either registry variant, which means that more relevant services are ranked before non-relevant ones in a result list of size 10. We have chosen this window size because we want to balance the number of retrieved candidates and the number of *relevant* retrieved candidates. We believe that a developer can certainly manually examine 10 Web Service descriptions without much effort. In concordance with related experiments that empirically evidence that users tend to select higher ranked search results [32], even a moderate improvement in this direction has a great impact on the discovery process. For instance, the probability that a user accesses the first ranked result is 90%, whereas the probability for accessing the second ranked one is, at most, 60% [32]. In this line, at least for these experiments, generating queries from DI-based code has shown to improve the discovery process. Moreover, the fact that the results related to DI-based code surpass those achieved by non-DI code regardless the variant of the registry used, provides empirical evidence that suggests that the improvements are explained by the usage of DI rather than the incidence of the underlying discovery mechanism.



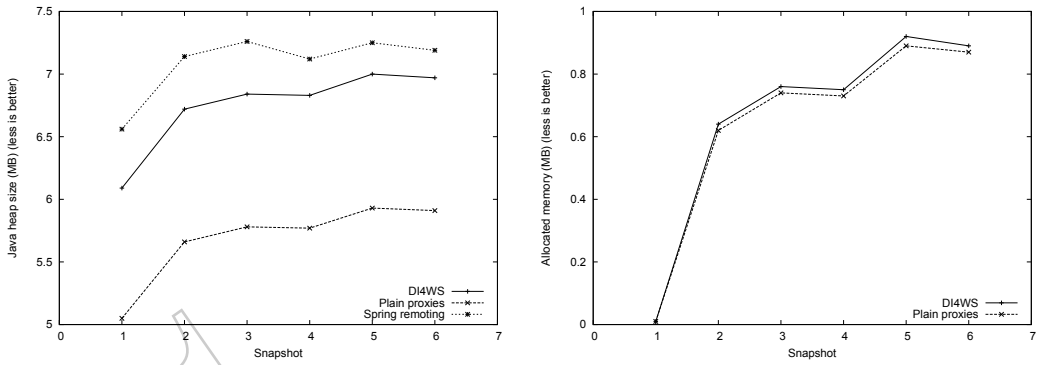
(a) Using the first variant of the registry (TF) (b) Using the second variant of the registry (TF-IDF)

Fig. 13. Average achieved results (%) for Recall-at-10, NR, R-Precision, Precision-at-1 and Precision-at-10 measures, when using commented source code

Alternatively, to test the hypothesis about comments positively impacting on the retrieval effectiveness, we recalculated the previous scenarios by taking into account the comments present in the source code files. Figure 13 shows the averaged results for each measure when using *commented* non-DI and DI-based code. Note that for all measures the commented DI-based code performed better than the commented and undocumented non-DI alternatives, and also the undocumented DI-based alternative. These experiments empirically confirm that mining comments from the source code of DI applications contributes to further improve the achieved effectiveness of discovery with each variant of the underlying service registry.

These positive results stem from the fact that DI-based code contains more meaningful terms than non-DI ones, therefore the former allows us to generate better queries. To illustrate this, let us return to the tables depicted in Fig. 4. Here, the non-DI query would consist of the 11 terms listed in the fifth column of the table associated with BookSearcher, this is, the internal component. Instead, the DI-based query would comprise 26=11+7+8 terms resulting from the union of the fifth column of the three tables. Intuitively, including more terms impacts not only on the service discovery effectiveness, but also on the processing time required to built and answer the corresponding queries. In other words, by increasing the input of the preprocessor (i.e. mining only commented dependant source code versus mining commented dependency code), we might expect the processing time required by the discovery process to increase.

To quantify the performance of service discovery, we measured the time required to build and answer queries by using the DI-based and non-DI versions of the above applications that had their source code commented. Tests were run on an Intel Core 2 Duo (1.83 GHz. per core) under JDK 1.6, and a service registry running on an Intel Pentium D (working at 3 GHz. and 1 GB of RAM) using JDK 1.6, Tomcat 5.5 and jUDDI 2.0rc5 [33]. Both machines were connected through a 100 Mbps LAN. To mitigate the noise, we averaged the elapsed time for 30 executions of the applications. Results showed that, at least for these applications and the aforementioned evaluation-set, the time required to solve an individual query was in the range of 1.1-1.6 sec and 1.3-2.1 sec when performing queries based on dependant code and



(a) Average allocated memory (b) Average memory overhead

Fig. 14. Proxying Web Services: Memory consumption as the number of invoked services increases

dependency code, respectively. This suggests that our techniques for mining DI-based code may certainly be useful in practice since they do not incur in excessive performance overheads.

Additionally, we measured the memory consumption overhead, that was caused by the utilization of the DI and Adapters patterns. By employing these patterns, new indirections are introduced in an application, because all the interactions between local components and outsourced ones are done through the *service adapters*. Then, we might expect the memory consumed by a DI4WS application to be greater than the memory required by a contract-first one. To measure memory consumption under DI4WS, we developed a simple application that accessed 10 of the Web Services of the evaluation-set, each having 3 operations on average. Furthermore, we obtained three variants for the application depending on the way these Web Services were consumed: plain proxies (generated with Eclipse WTP 3.2.2), Spring remoting, and DI4WS (generated by running our plug-in under Eclipse WTP 3.2.2). All tests were run under JDK 1.6. To perform a fair comparison, in all cases, the application components were coded by using DI and assembled via the Spring framework. To perform the measurements, we used *jhat*, an utility that comes with the JVM 1.6 and allows programmers to dump the entire object graph of a running Java application.

Figure 14 (a) shows the average amount of allocated memory for 30 runs of the applications. As depicted, we took memory snapshots upon initializing the JVM and the Spring run-times (i.e. when the application components and services are wired together; snapshot 1) and after calling 2, 4, 6, 8 and 10 Web Services (snapshots 2 through 6). As illustrated in Fig. 14 (a), the variants initially allocated very different amounts of RAM, because they are subject to quite different DI configurations.

The memory allocation curves of the three applications behaved similarly. The variant using Spring remoting incurred in an overhead in the range of 22-30% and 3-8% with respect to the variant using plain proxies and DI4WS, respectively. The cause of the overhead is that, unlike the other two alternatives, Spring remoting uses class introspection and aspecting to proxy Web Services at run-time, which results in more allocated objects. Figure 14 (b) shows the memory overhead of the DI4WS implementation with respect to the variant using plain proxies, without taking into account the allocated memory upon initializing the JVM. All in

all, maintaining a service adapter requires an average of 2% of extra memory compared to maintaining a plain proxy. Actually, this value is smaller, because at run-time DI4WS adapters share many objects such as factories and internal structures of the Spring framework, but whose size is very difficult to measure in a precise way. Nevertheless, we believe this overhead is acceptable since DI4WS produces cleaner code, enables for looser coupling and simplifies service discovery.

6 Related work

Currently, lightweight frameworks for developing complex enterprise systems promote the use of DI as a way to improve the quality of their resulting designs, mainly in terms of extensibility, testability and reusability [34]. Frameworks such as PicoContainer [35], Spring [36] and HiveMind [37] have become highly popular among Web developers. Besides, the notions behind DI have been already bundled into existing methods and tools for modeling Web applications, such as FrameWeb [38]. Despite the hype of DI, as far as we know, there is in the literature one empirical assessment of the impact of DI on Web-based projects, one article on the use of the pattern in Grid development, and one study of the adoption of DI by the software industry.

First, [15] analyzes 20 sets of Web-based projects, in which there is, at least, one project using DI per set. The authors recollect and compare classic measures related to the type and strength of relationships among the components of both DI and non-DI applications. The analysis focused on quantifying how DI impacts on software maintenance and concluded that software tend to have lower coupling when using DI.

Second, [39] presents a DI-based programming model to develop applications for service-oriented Grids [40]. The authors qualitatively analyze the outcomes of employing DI in such a distributed environment by Grid-enabling two applications by using and not using DI. Then, classic maintainability metrics are taken on the resulting codes. The experiment showed that the DI-based code was lighter than the non-DI version, suggesting reduced maintainability cost and development effort.

Finally, [16] presents a benchmark for determining whether the design of a software system is capable of using DI or not. The benchmark is essentially based on two types of elements, namely operational definitions for the DI pattern and code analysis techniques for detecting its use. The benchmark is evaluated by using 34 open source applications, and concludes that, surprisingly, there is not strong evidence that suggest a widespread use of DI.

To sum up, at the time of writing this paper, there are empirical evidences to suggest that using DI improves software maintainability through abstraction and indirection mechanisms. Nevertheless, intuitively, the more the indirections, the higher the demands for memory and CPU. In this sense, experimental evidences about the overhead imposed by using DI have not been rigorously analyzed until now. Moreover, in the context of SOC development, DI has not been completely exploited yet.

In spite of the lack of studies about the practical implications of using DI in SOC, which in fact aligns with the absence of studies on the costs and benefits of the adoption of the SOC paradigm itself [41], there are however a substantial body of work that to a lesser or greater extent share some of the ideas of the approach to service outsourcing followed in this paper. Specifically, the idea of seamless "injection" of Web Services into client applications is

also promoted by the Web Services Management Layer (WSML) [42]. Conceptually, WSML introduces a software layer similar to the adapter layer of DI4WS that isolates applications from concrete service providers by using aspect-oriented techniques. Though the authors have meticulously discussed and analyzed WSML, the soundness of the approach has not been corroborated experimentally yet. [43] also uses aspect orientation to dynamically replace an internal application method by a similar Web Service operation. Unlike DI4WS, the approach aims at fully automating the tasks of discovery and consumption of services at run-time, which has historically received criticism [44] as it is difficult to perform Web Service engagement without any user intervention through the current support provided by UDDI and WSDL.

Moreover, the use of service adapters or similar mechanisms to isolate application logic from Web Service concerns is explored in several works, particularly for semi-automatic generation of service representatives or proxies [45], coordination and compensation of Web Service transactions [46], and non-invasive in-the-middle data transformations to interact with data services [47]. Unlike these approaches, DI4WS takes advantage of both the implicit semantics of the internal components of the application logic and potential interfaces of the required external services to generate adapter code and to discover appropriate Web Services. With respect to service discovery, close to our work is that of Dourdas et al. [19] and Blake et al. [18] who also investigated the automatic building of queries from software artifacts for discovering services. Basically, the former work relies on mining textual features from natural language descriptions of service requirements and using them to query a service registry. The latter work is a personal software agent that gathers keywords from developers' IDE (e.g. the name of the project, the developer's role, etc.) for assisting developers in finding services. Essentially, approaches aimed at exploiting the *implicit* semantics of client-side applications to assist developers in the process of discovering services represent an alternative to semantic discovery, which has shown to be very difficult to use in practice [7]. The main difference between the aforementioned works and DI4WS is that since this latter encourages developers to build their service-oriented applications by employing a well-known design pattern (i.e. DI), DI4WS can gather descriptive keywords from the application while exploiting the structure of the pattern and best practices for naming and documenting source code.

7 Conclusions

In this article, we have studied the role of the Dependency Injection pattern in the context of service oriented applications. Concretely, the study was conducted in the context of DI4WS, an approach for outsourcing Web Services based on the benefits for building maintainable, loosely-coupled components inherent to DI, and provides a development model for making the task of building SOC applications easier.

The utmost goal of DI4WS is to exploit the information present in the client-side code to ease the task of discovering Web Services, and at the same time let programmers to separate the application logic from service-specific aspects to increase the maintainability of the resulting software. Experiments suggest that DI4WS is beneficial from a practical perspective, as it allows developers to quickly and seamlessly outsource services, with a minimal overhead in terms of system resources. Despite these encouraging results, we will conduct more experiments to further validate DI4WS. Opportunities for future evaluation include using other Web Service collections and more applications. To this end, we are planning to employ a recently

published repository of real Web Services[k] to conduct controlled case studies with several development teams. This will allow us to further evaluate DI4WS as well as to use specialized software metrics to better assess the quality of the produced software and to consider human factors.

Another line of future research involves the provision of assistance to developers for coding service adapters. Concretely, we are planning to extend the technique proposed in [45] to partially automate the task of bridging the signatures of the methods implemented by a service adapter and the operations of its associated service proxy. This technique will identify structural differences between two interfaces (e.g. parameter types/ordering, missing/extra parameters, etc.) and semi automatically generates bridging code accordingly. Then, the mismatches that require developer's input for their resolution could be presented graphically to the user through our plug-in.

As DI4WS uses a centralized, client-server service matchmaking and discovery scheme that may not be suitable for large SOC environments, we are working on scalable solutions to address this issue. We are currently developing parallel and distributed versions of our text mining algorithms on top of JGRIM [39], a middleware that supports efficient execution of component-based Java applications on computational Grids.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper.

References

1. Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.
2. Martin Bichler and Kwei-Jay Lin. Service-oriented computing. *IEEE Computer*, 39(3):99–101, 2006.
3. Steven J. Vaughan-Nichols. Web Services: Beyond the hype. *Computer*, 35(2):18–21, 2002.
4. Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003.
5. W3C Consortium. WSDL version 2.0 part 1: Core language. W3C Candidate Recommendation, `http://www.w3.org/TR/wsdl20`, June 2007.
6. OASIS Consortium. UDDI version 3.0.2. UDDI Spec Technical Committee Draft, `http://uddi.org/pubs/uddi_v3.htm`, October 2004.
7. Rob McCool. Rethinking the Semantic Web. part I. *IEEE Internet Computing*, 9(6):88, 86–87, 2005.
8. Hongbing Wang, Joshua Zhexue Huang, Yuzhong Qu, and Junyuan Xie. Web Services: Problems and future directions. *Journal of Web Semantics*, 1(3):309–320, 2004.
9. John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis. Contemporary Web Service discovery mechanisms. *Journal of Web Engineering*, 5(3):265–290, 2006.
10. Marco Crasso, Alejandro Zunino, and Marcelo Campo. Easy Web Service discovery: a Query-by-Example approach. *Science of Computer Programming*, 71(2):144–164, 2008.
11. Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. Web Services Invocation Framework (WSIF). In *Workshop on Object-Oriented Web Services (OOWS '01) - ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01), Tampa, Florida, USA*, New York, NY, USA, 2001. ACM Press.

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

12. Apache Software Foundation. Apache CXF: An Open Source Service Framework. `http://cxf.apache.org`, 2009.
13. Massimo Paolucci and Katia Sycara. Autonomous semantic Web Services. *IEEE Internet Computing*, 7(5):34–41, 2003.
14. Rod Johnson. J2EE development frameworks. *Computer*, 38(1):107–110, 2005.
15. Ekaterina Razina and David Janzen. Effects of Dependency Injection on maintainability. In *11th IASTED International Conference on Software Engineering and Applications (SEA '07), Cambridge, MA, USA*, Calgary, AB, Canada, 2007. ACTA Press.
16. Hong Yul Yang, E. Tempero, and H. Melton. An empirical study into use of dependency injection in java. In Liang-Jie Zhang, editor, *19th Australian Conference on Software Engineering (ASWEC '08)*, pages 239–247, Perth, WA, USA, March 2008. IEEE Computer Society.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.
18. M. Brian Blake, Daniel R. Kahan, and Michael Fitzgerald Nowlan. Context-aware agents for user-oriented Web Services discovery and execution. *Distributed and Parallel Databases*, 21(1):39–58, 2007.
19. Nektarios Dourdas, Xiaohong Zhu, Neil Maiden, Sara Jones, and Konstantinos Zachos. Discovering remote software services that satisfy requirements: Patterns for query reformulation. In *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag, 2006.
20. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA, USA, 2002.
21. Mike P. Papazoglou and Willem-Jan Heuvel. Service Oriented Architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.
22. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.
23. Robert R. Korfhage. *Information Storage and Retrieval*. John Wiley & Sons, New York, NY, USA, 1997.
24. Gerad Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
25. Martin F. Porter. An algorithm for suffix stripping. In *Readings in Information Retrieval*, pages 313–316. Morgan-Kaufmann Publishers, San Francisco, CA, USA, 1997.
26. Padmal Vitharana, Hemant Jain, and Fatemeh Zahedi. Strategy-based design of reusable business components. *IEEE Transactions on Systems, Man, and Cybernetics*, 34(4):460–474, November 2004.
27. Parag C. Pendharkar and James A. Rodger. An empirical study of factors impacting the size of object-oriented component code documentation. In *20th annual international Conference on Computer Documentation (SIGDOC '02), Toronto, Ontario, Canada*, pages 152–156, New York, NY, USA, 2002. ACM Press.
28. Diomidis Spinellis. The way we program. *IEEE Software*, 25(4):89–91, 2008.
29. Eclipse Foundation. Eclipse Java Development Tools (JDT). `http://www.eclipse.org/jdt`, 2009.
30. Eclipse Foundation. Web Tools Platform (WTP) Project. `http://www.eclipse.org/webtools`, 2009.
31. Andreas Heß, Eddie Johnston, and Nicholas Kushmerick. ASSAM: A tool for semi-automatically annotating semantic Web Services. In *The Semantic Web - ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2004.
32. Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning user interaction models for predicting web search result preferences. In *29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06), Seattle, Washington, USA*, pages 3–10, New York, NY, USA, 2006. ACM Press.
33. Apache Software Foundation. jUDDI. `http://ws.apache.org/juddi`, 2009.
34. Rod Johnson and Juergen Hoeller. *Expert One-on-One J2EE Development without EJB*. John

Wiley & Sons, 2004.
35. PicoContainer Committers. PicoContainer. `http://www.picocontainer.org`, 2008.
36. SpringSource. The Spring Framework. `http://www.springsource.org`, 2009.
37. Apache Software Foundation. HiveMind. `http://hivemind.apache.org`, 2009.
38. Vítor Estêv ao Silva Souza and Ricardo de Almeida Falbo. FrameWeb: A framework-based design method for web engineering. In *2007 Euro American Conference on Telematics and Information Systems (EATIS '07), Faro, Portugal*, pages 1–8, New York, NY, USA, 2007. ACM Press.
39. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Grid-Enabling Applications with JGRIM. *International Journal of Grid and High Performance Computing*, 1(3):52–72, 2009.
40. Malcolm Atkinson, David DeRoure, Alistair Dunlop, Geoffrey Fox, Peter Henderson, Tony Hey, Norman Paton, Steven Newhouse, Savas Parastatidis, Anne Trefethen, Paul Watson, and Jim Webber. Web Service Grids: An evolutionary approach. *Concurrency and Computation: Practice and Experience*, 17(2-4):377–389, 2005.
41. John Erickson and Keng Siau. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of Database Management*, 19(3):42–54, 2008.
42. María Agustina Cibrán, Bart Verheecke, Wim Vanderperren, Davy Suvée, and Viviane Jonckers. Aspect-oriented programming for dynamic Web Service selection, integration and management. *World Wide Web*, 10(3):211–242, 2007.
43. Marisol Pérez Reséndiz and José Oscar Olmedo Aguirre. Dynamic invocation of Web Services by using aspect-oriented programming. *2nd International Conference on Electrical and Electronics Engineering*, pages 48–51, 2005.
44. Shuping Ran. A model for Web Services discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, 2003.
45. Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web (WWW '07), Banff, Alberta, Canada*, pages 993–1002, New York, NY, USA, 2007. ACM Press.
46. Michael Schäfer, Peter Dolog, and Wolfgang Nejdl. An environment for flexible advanced compensations of Web Service transactions. *ACM Transactions on the Web*, 2(2):1–36, 2008.
47. Ioana Manolescu, Marco Brambilla, Stefano Ceri, Sara Comai, and Piero Fraternali. Model-driven design and deployment of service-enabled Web applications. *ACM Transactions on Internet Technology*, 5(3):439–479, 2005.