Modularizing Client-Side Web Service Management Aspects

María Agustina Cibrán[1], Bart Verheecke[1], Viviane Jonckers

{Maria.Cibran; Bart.Verheecke}@vub.ac.be, vejoncke@info.vub.ac.be

System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium

Abstract. In the short time Web Services have been around, a lot of development tools became available to create and deploy web services. However, typical approaches for the integration of web services require the client applications to hard-wire their references. To improve maintainability and achieve high flexibility in the integration of services we proposed in previous work an intermediate abstraction layer, called Web Services Management Layer (WSML). The WSML allows decoupling services from the application, selecting services depending on specific criteria and monitoring and managing additional service concerns. In this paper we extend that work and present the results of a case study where we implemented the service redirection mechanism extended with a local and global cache. We show how dynamic AOP, and in particularly the JAsCo language, is ideally suited to implement the core functionality of the WSML.

1. Introduction

Web Services are modular applications that are published, localised and invoked over a network using XML-standards like SOAP, WSDL and UDDI. The aim of Web Service technology is to allow easy integration of different business processes regardless of the software or hardware used. It promises a platform independent solution to wire distributed applications of different enterprises. Although an impressive range of development tools enable the creation, deployment and management of Web Services at the server-side, not many approaches exist to facilitate the integration and management of services at the client side. Moreover current approaches for the integration of Web Services are rather static. For instance, in the **Wrapper Approach** each Web Service is wrapped and treated as an internal software component. As a consequence, by hard-wiring Web Services references in the client applications, the specific requirements of services are completely ignored.

[1] Funded by the IWT (Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen), Mosaic Project, Flanders (Belgium)

Another disadvantage is that to deal with potential service failures and other management concerns, the applications have to include extra code which is scattered and tangled with the application code. This approach is adopted in many state-of-the-art tools like MS Visual Studio.NET [1] and BEA WebLogic [2]. A more flexible and dynamic approach for the integration of Web Services is the one of **tModels** [3]. Using this solution the applications do not include specific services but tMoldes that represent templates of the functionality required. However only services that exactly implement the system's tModel specification can be dynamically integrated while services with other interfaces and service compositions that deliver the same functionality cannot be straightforwardly included. Moreover, management code still needs to be written manually and remains scattered over the application code.

We propose an intermediate layer called **Web Services Management Layer (WSML)** [4] that focuses on achieving *dynamic and flexible client-side integration and management* of services. The WSML is placed in between the client application and the world of Web Services. Its main objective is to decouple the client application from specific Web Services references, enhancing robustness and maintainability. By weakening the link between the applications and the services, hot-swapping functionality based on service availability can be installed. Additionally, the WSML can realise a more advanced selection mechanism which can also contemplate service criteria based on non-functional properties of services and deal with other client-side management concerns. In [4] we demonstrated how dynamic *Aspect Oriented Programming* (AOP) [5] [6] and in particular how the dynamic AOP language called JAsCo [7] [8] is ideally suited to build the core functionality of this management layer.

In this paper we concentrate on the mechanism used to loose the link between applications and the services based on JAsCo aspects and connectors and enhance it with local and global caching functionality. By caching we mean a mechanism to store the results of Web Service invocations in order to be reused in future invocations. The objective is to reduce the number of expensive remote web calls to services or to provide a result to the client even in the case no services are available at the moment of the request. We show how JAsCo aspects and connectors are also suitable to plug global and local caching functionality as part of the redirection mechanism.

The next section gives a brief to introduction the functionality of the WSML. Section 3 presents an overview of Aspect Oriented Programming and JAsCo's features. Our solution is presented in section 4. In section 5, related and future work is presented. Finally, we conclude in section 6.

2. WSML

To facilitate the integration of Web Services in a client application we presented in [4] the **Web Services Management Layer (WSML)**, which allows the realization of

the *just-in-time integration of services and service compositions*. When a client application requests certain service functionality, it refers to a **Service Type**[2] , a generic specification of the required functionality without any reference to concrete services. A service type allows hiding syntactical differences between semantically equivalent services. When the application makes a *request* on a service type, the WSML is responsible for making the translation to concrete service invocations and returning the result to the application. This *dynamic binding* makes the application more adaptive to changes in the environment as services can be easily added and removed. Also, the application becomes more robust as it does not depend on a single service anymore. If a single service or service composition fails or becomes unreachable, the WSML can easily switch to another one.

Furthermore, our solution is more flexible as it not only provides a dynamic, adaptive way of integrating services, but also allows to encapsulate several *client-side* service management issues that otherwise would remain scattered and tangled in the client application. Examples of management concerns are billing, accounting, transaction, security, caching, logging, etc. In section 4 of this paper, concrete examples of billing and caching are demonstrated.

The WSML can run as an extended part of the client application (i.e. on the same virtual machine) or it can be deployed on a different machine in an application independent way. The WSML can look up available services in a central UDDI-register [9] or discover them using decentralised Web Services Inspection Language (WSIL) documents [10]. The decision of which service will effectively be selected is done at run time considering the available services and taking into account non-functional requirements pursued by the application (e.g. price and speed requirements). In the next section we introduce dynamic AOP, the technology adopted to realize the core functionality of the WSML.

3. Aspect Oriented Programming

Aspect Oriented Programming (AOP) [5] [6] argues that some concerns of a system cannot be cleanly modularized using current software engineering methodologies. Examples of crosscutting concerns are synchronisation and logging. As a result they remain scattered over different modules of the system and even tangled with code that addresses other concerns. Due to this code duplication, it becomes very hard to add, edit and remove such a crosscutting aspect in the system. The goal of AOP is to achieve a better separation of concerns by introducing a new concept to modularize crosscutting concerns, called *aspect*. An aspect defines a set of *join points*, places in the target application where the normal execution is altered. *Aspect weavers* are used to weave the aspect logic into the target application. Nowadays, several AOP approaches, such as AspectJ, Composition Filters [11], HyperJ [12] and DemeterJ [13] are available and being applied in research and industrial projects.

---

[2] Called **Abstract Service Interfaces (ASI)** in previous work.

As shown in [4] aspects are suitable to avoid tangling the application code with service related code. Moreover, it is desired to plug in and out service selection and management concerns at run time, due to the dynamic nature of the Web Service environment. To this end, the dynamic AOP language JAsCo [7] [8] is ideal since it provides support to plug-in and out aspects at run-time. JAsCo is built on top of Java and it defines two new concepts:

- **Aspect Beans:** specify crosscutting behaviour in a reusable manner, independently of any concrete deployment. It can define one or more logically related hooks as a special kind of inner classes. A hook specifies ***when*** the normal execution of a method should be intercepted and ***what*** extra behaviour should be executed at those points. This extra behaviour can be placed before, after or in replacement of the original behaviour.
- **Connectors:** deploy the crosscutting behaviour of the aspect beans in a concrete context. It specifies ***where*** the crosscutting behaviour should be deployed. Connectors can include the definition of combination strategies to specify how the behaviours specified in different hooks have to be combined. Connectors can be dynamically created and removed.

4. AOP for Web Service Management

In this section we present our approach for the redirection of abstract requests to concrete services and Web Service management, which is based on JAsCo aspects and connectors. In section 4.1 we explain the basic redirection mechanism and demonstrate how it allows hot swapping between services. An example of a management aspect that deals with client-side billing is shown in section 4.2. Next, we introduce a more advanced redirection mechanism using global caching in section 4.3 and local caching in section 4.4. These examples demonstrate the suitability of the adopted dynamic AOP technique for achieving the decoupling and encapsulation of the service management concerns as part of the WSML.

4.1 Basic Redirection Mechanism and Hot-Swapping

The client application will make request calls to a Service Type instead of a concrete service. To realise dynamic binding we introduced a redirection mechanism based on JAsCo aspects and connectors in [4]. In the rest of this section, we present an improved version of this approach. However, first we introduce a running example. Consider a holiday booking application that wants to include the functionality of a third party hotel service to request and book hotels. Therefore, a *HotelServiceType* is specified with a web method *listHotels* which returns a list of available hotels for a given period and destination and a method *bookHotel* which books a specific hotel for a given period. To enable the redirection of requests on the *HotelServiceType* to concrete hotel services, we introduce *Service Redirection Aspects*. They specify that each time some functionality is requested in the application (***when***) this is replaced by

the invocation of a concrete method on a Web Service (***what***). This replacement behaviour is specified in aspect hooks; there is a hook for each request the Web Service can fulfil. Next, a connector is used to provide context and to specify exactly when the replacement behaviour of the aspect hooks should be executed (***where***), in this case, each time the methods *listHotels* and *bookHotel* are called in the application. As multiple services can be available to fulfil the functionality of the request, multiple connectors will be present (one for each service redirection aspect). Now, we can simply realise hot-swapping by activating and deactivating these connectors as depicted in figure 1. In this case, *WebServiceC* is used to fulfil the request *listHotels* because its corresponding (grey) connector is activated. Only the replacement behaviour of *Service C redirection aspect* is executed, and therefore only *WebServiceC* is addressed.

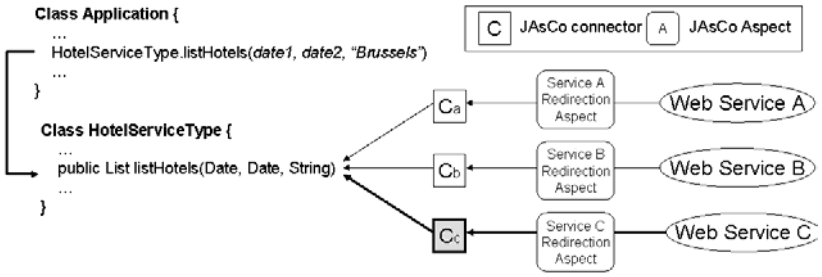

Fig. 1. The HotelServiceType with three available web services

The mechanism presented here is very flexible as new services can be plugged into the WSML by simply creating new connectors and service redirection aspects that encapsulate the service invocation details. A *Selection Module* in the WSML is responsible for choosing which connector should be activated at a given time. This is done based on the *availability status* of the service. If a service is not reachable due to the network conditions or service related problems, automatically another connector is activated. To this end, the WSML checks with fixed intervals if all services are still reachable. Furthermore, notifications can be sent to the WSML about the availability status of services. For instance, an external UDDI-registry or a service provider can send a message to inform about the down time of its service. This means that the WSML can pro-actively select the most optimal connector, and therefore minimize the impact on the application.

Figure 2 shows the code of the *WebServiceC_RedirectionAspect* in JAsCo. The aspect contains two hooks: one for the listHotels request (lines 4 to 13) and one for the bookHotel request (lines 14 to 16). In the first hook the actual web method to list the available hotels is invoked in line 8. If the call to a service fails, a *fallback mechanism* is executed in the catch statement: first, the WSML is informed of the failure in line 11, which will result in the deactivation of the connector of *WebServiceC* and the activation of another connector, one of the functionally equivalent web services A or B. To execute the behaviour of this new connector, the original method is called again in line 12.

```
1.  Class ServiceC_RedirectionAspect {
2.    Stub stub;  //client stub of WebServiceC
3.     …
4.    hook ListHotelsHook {
5.      ListHotelsHook(method(..args)){ execute(method); }
6.        replace() {
7.          try {
8.            return stub.listAvailableHotels (args);
9.          }
10.         catch (ServiceUnavailableException e) {
11.           WSML.failureNotification ("WebServiceC");
12.           return method(args);
13.         }}}
14.   hook BookHotelHook {
15.     …
16.   }}
```

Fig. 2. The redirection aspect for WebServiceC

The connector for *WebServiceC* is shown in figure 3. The hooks defined in the redirection aspect are instantiated and linked to the appropriate methods of the Service Type. The actual replacement behaviour is executed in lines 6 and 7.

```
1.  static connector ServiceC_RedirectionConnector {
2.    ServiceC_RedirectionAspect.ListHotelsHook listHook =
3.      new ServiceC_RedirectionAspect.ListHotelsHook
4.        (List HotelServiceType.listHotels (Date, Date, String));
5.    ServiceC_RedirectionAspect.bookHotelHook bookhook = …
6.    listHook.replace();
7.    bookhook.replace();
8.  }
```

Fig. 3. The redirection connector for WebServiceC

4.2 Billing Mechanism

Now, suppose the service provider of *WebServiceC* states that every time the method *listHotels* is invoked on its service, a certain amount needs to be paid. As a controlling mechanism for the client application, the WSML can monitor the payment information by simply adding a billing aspect that implements a billing per use schema. This is depicted in figure 4. The first hook does the actual calculation of the total amount; the second hook is used to update the attribute *cost* defined in the aspect whenever the cost of the service changes.

```
1.  class BillingPerUseAspect {
2.    private String serviceName;
3.    private int cost, total;
4.    …
5.    hook BillHook {
6.      BillHook(method(..args)) {execute(method);}
7.      after() {total = total + cost;}
8.    }
```

```
9.    hook UpdateCostHook {
10.     UpdateHook(method (..args)) {execute(method);}
11.   after() {
12.     //retrieve newCost from service
13.     cost = newCost;
14.   }
15. }
```

Fig. 4. An BillingPerUseAspect that implements a billing per use schema

To actual do the calculation of billing when the method *listHotels* is executed, the redirection connector of figure 3 is extended with the code of figure 5 to initialise and call the replace behaviour method of the *PayHook*. Code to initialise the *UpdateCostHook* is not shown here, but it will typically hook into the moment a change in the policy of the service is detected (for example when the provider of *WebServiceC* changes the description file of its service) or when the WSML receives a message that announces the changing of the price.

```
1.  static connector ServiceC_RedirectionConnector {
2.    ...continued from line 7 of figure 3
3.    BillingPerUseAspect.BillHook billPerUseHook =
4.      new BillingPerUseAspect.BillHook
5.       (List HotelServiceType.listHotels(Date, Date, String));
6.    billPerUseHook.after();
7.  }
```

Fig. 5. Extension of the redirection connector for the BillHook of the BillingPerUseAspect

4.3 Global Caching Mechanism

The redirection mechanism introduced in section 4.1 can be extended with a global caching mechanism. To avoid that unnecessary calls are being made to services, the WSML can store the results from previous service invocations and provide them to the client application when needed. This can be achieved by introducing a *Caching Redirection Aspect.* It works in a similar way as the service redirection aspect, except it fetches its results from a cache database instead of a web service.

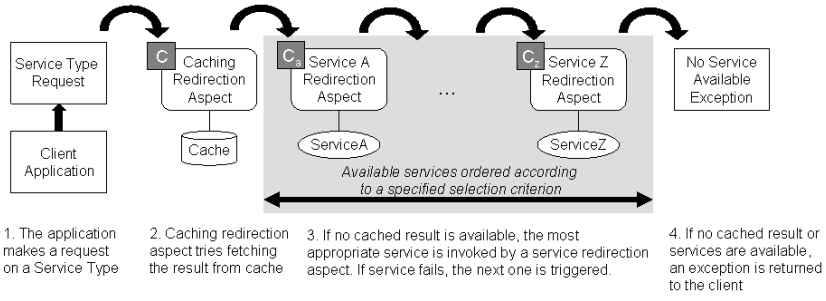

Fig. 6. The redirection mechanism extended with global caching

Figure 6 depicts the flow of execution from the moment the client application makes a request on a service type. The Caching Redirection Aspect hooks into this moment and executes its replacement behaviour, i.e. trying to retrieve a result from the cache. If this fails, the selection module is informed to activate the connector of the most appropriate service. This is done based on some specified selection criterion (e.g.: select the cheapest or fastest service available). This mechanism can also be achieved in a similar way by using service selection aspects and monitoring aspects. If for some unforeseen reason the selected service fails, the next one is invoked by activating the next connector as already described in section 4.1. If a result is acquired from a service, it is stored in the cache and returned to the application. Note that there is one cache *for all services*, hence the name *global* caching. In figure 7 the code of the Caching Redirection Aspect is shown.

```
1.  Class CachingRedirectionAspect {
2.    Type result;
3.    hook CachingHook {
4.      CachingHook (method(..args)){
5.        execute(method);
6.      }
7.    isApplicable() {
8.      result = db.getLastResult(args);
9.      return result.valid();
10.   }
11.   replace() {
12.     if (result.valid())
13.       return result;
14.     else {
15.       WSML.selectBestService();
16.       result = calledObject.method(args);
17.       db.storeResult (result);
18.       return result;
19.     }
20.   }
21. }
```

Fig. 7. Caching Redirection Aspect

The connector for this aspect is analogue to the connectors for the service redirection aspects and is shown in figure 8. Note that the billing mechanism as described in the previous section still behaves correctly with this cache installed. Only when an actual service invocation is done, the total cost is updated.

```
1.  static connector CachingRedirectionConnector {
2.    CachingRedirectionAspect.CachingHook cachingHook =
3.      new CachingRedirectionAspect.CachingHook
4.        (List HotelServiceType.listHotels (int, int, String));
5.    cachingHook.replace();
6.  }
```

Fig. 8. Caching Redirection Connector

4.4 Local Caching Mechanism

With only a small alteration of the code we can also implement a *local* caching mechanism. By local caching we mean that *for each service*, the result of a concrete invocation is cached so that it can be reused for future requests of the same functionality.

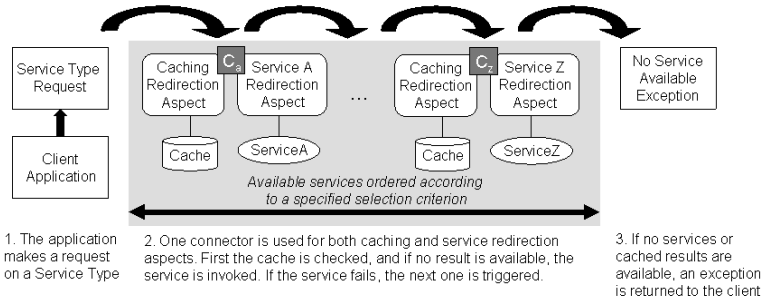

Fig. 9 The redirection mechanism extended with local caching

Figure 9 shows how the local caching works: each time the client application requests some service functionality, the selection module decides which service is the most appropriate to fulfil that requirement and activates the corresponding connector. In that connector it is specified that before the real invocation of the service, the caching mechanism is triggered. As a consequence, if there is a cached result that corresponds to that requirement, that value is returned and the service is not invoked. If there is no valid cashed value for the request, then the web service invocation is performed and the cache is updated with the new result. The code of the Caching Redirection Aspect is the same as for the global caching (see figure 7). The code for the "shared" connector *ServiceC_RedirectionConnector* is depicted in figure 10. A precedence strategy is used to denote that first the replacement behaviour of the *cachingHook* is executed.

```
1.  static connector ServiceC_RedirectionConnector{
2.    ... continued from line 6 of figure 5
3.    CachingAspect.CachingHook cachingHook =
4.      new CachingAspect.CachingHook
5.        (List HotelServiceType.listHotels(Date, Date, String));
6.
7.    ExcludeCombinationStrategy excludeBilling = new
8.     ExcludeCombinationStrategy(cachingHook,billPerUseHook);
9.    addCombinationStrategy(excludeBilling);
10.                                          Combination strategy
11.   cachingHook.replace();
12.   listHotelsHook.replace();
13.   billPerUseHook.after();
14. }                                        Precedence strategy
```

Fig. 10. Redirection connector with support for local caching

To ensure that the billing is not performed when the result is retrieved from the cache, an exclude combination strategy is specified (lines 7 to 9). This combination strategy ensures that the billing hook is excluded when the caching hook is applicable. As a result the billing is done only when the concrete web service is invoked and not when the result is returned from the cache.

5. Related and Future Work

Dynamic binding also exists in other middleware technologies such as CORBA [14]. High level abstractions of objects are provided in CORBA by interfaces specified in the Interface Definition Language (IDL). These interfaces can be used either statically or dynamically. An interface is statically bound to an object when the name of the object it is accessing is known at compile time. In addition, clients that need to discover an object at run time and construct a request dynamically can use the Dynamic Invocation Interface (DII). By accessing information in the interface repository, a client can retrieve all of the information necessary about an objects interface to construct and issue a request at run time. DII essentially saves the programmer from the trouble of having to generate the necessary stubs and skeletons. The client application is still written to the specific interface and the meaning of specific methods of the remote object is embodied in the client code. The WSML is a more adaptive approach because it is able to use any interface that provides the requested functionality, in a completely transparent way for the client.

Other relevant approaches to deal with integration and management of third-party web services are: The Web Service Description Framework (WSDF) [15] incorporates ideas from the Semantic Web community [16] suggesting an ontological approach for the side effect free invocations of services. Syntactical differences between services are stored in a database. The Web Services Mediator (WSM) [17] also identifies the need for a mediation layer to achieve dynamic integration of services. However, as far as we know there is no implementation available of the WSM. The Web Services Invocation Framework (WSIF) [18] supports a Java API for invoking web services irrespective of how and where the services are provided. WSIF mostly focuses on making the client unaware of service migrations and change of protocols. Our approach also allows dealing with service changes as all communication details of services are hidden inside separate aspects as explained in section 4.1. We are currently investigating how the WSML can encapsulate in a modular way various other communication details that deal with web service protocols such as WS-Security [19], WS-Policy [20], WS-Transaction [21] and business process definition languages such as BPEL4WS [22].

The idea of applying AOP concepts to decouple web services concerns is quite innovative and thus not many approaches have been developed focusing on this field. However, Arsanjani et al. [23] have recently identified the suitability of AOSD to modularize the heterogeneous concerns involved in web services. However they refer

to approaches like Aspect/J [24] and the Hyper/J [25] [26] which, contrary to JAsCo only allow static aspect weaving.

We also use aspects for other purposes in the WSML: at the moment we are working on realising a more advanced hot swapping mechanism which not only selects services based on availability but also can consider service policies based on non-functional properties, such as price, speed, distance, reliability, etc. For this, we use *selection aspects* that encapsulate these policies and *monitoring aspects* that set up measuring points to gather the needed information to make the selection decisions.

Part of our ongoing research is to achieve dynamic discovery of services and to determine the compatibility between Service Types and available services and how the WSML should make the mapping between the two. For this, we are currently investigating several ontology approaches, including DAML-S [27] and Meteor-S [28].

In the same way the caching concern was addressed as part of the WSML, other management issues can be tackled, like billing accounting, security, transaction management, etc. We are currently working on the definition of a library of reusable aspects that would allow the application developer to dynamically instantiate and configure the needed aspects to deal with different service management concerns. These reusable aspects can be seen as generic templates that can be customized and integrated "on demand" to accommodate to service requirements.

6. Conclusions

This paper focuses on the mechanism implemented in the WSML to redirect application requests to web services invocations and suggests an enhanced version of it which incorporates functionality to cache service invocations results. This mechanism allows client applications to request service functionality without having to commit to specific web services. Instead, applications request the needed functionality to service types, being the WSML responsible of mapping those requests to concrete web services invocations. JAsCo aspects and connectors are used to implement the redirection mechanism. As a consequence of the dynamic addition and removal of JAsCo connectors, hot swapping of services is realised.

Two types of caching are addressed which depend on the moment when the caching takes place: global caching occurs before the selection of the most appropriate service for a given request. Local caching takes place after the most appropriate service for the given request has been selected and before it is effectively invoked. We have presented a solution for both cases based on JAsCo aspects and connectors. The dynamic nature of JAsCo allows to dynamically and non-invasively plug-in and out the caching concern into the system, without having to modify the client application.

7. References

[1] Microsoft, "XML Web services," Visual Studio.NET Technical Resources, http://msdn.microsoft.com/vstudio/techinfo/articles/xmlwebservices/default.asp
[2] Bea WebLogic Platform, Bea Systems, http://www.bea.com
[3] Microsoft, "Introduction to UDDI Services and tModels," 2002, http://isb.oio.dk/uddi/help/en/intro.whatisuddi.aspx
[4] B. Verheecke, M. A. Cibrán, V. Jonckers, "AOP for Dynamic Configuration and Management of Web Services," Proc. of ICWS'03-Europe, Germany, 2003.
[5] Aspect-Oriented Software Development. http://www.aosd.net/
[6] Communications of the ACM. Aspect-Oriented Software Development, October 2001.
[7] D. Suvée and W. Vanderperren. "JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development". Proc. of Int. Conf. on AOSD, USA, 2003.
[8] W. Vanderperren, D. Suvée, B. Wydaeghe and V. Jonckers. "PacoSuite & JAsCo: A visual component composition environment with advanced aspect separation features", Proc. of Int. Conf. on FASE, Poland, 2003.
[9] Uddi.org, "Universal Description, Discovery and Integration," UDDI Executive Whitepaper, 2001
[10] T. Appnel, "An Introduction to WSIL, the Web Services Inspection Language," O'Reilly on Java.com, 2002, http://www.onjava.com/pub/a/onjava/2002/10/16/wsil.html
[11] L. Bergmans, M. Aksit, "Composing Crosscutting Concerns Using Composition Filters", Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
[12] H. Ossher, P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software", Communications of the ACM, 44(10):43-50, October 2001.
[13] K. Lieberherr, D. Orleans, J. Ovlinger, "Aspect-oriented programming with adaptive methods", Communications of the ACM, Vol. 44, No. 10, pp. 39-41, October 2001.
[14] CORBA, Object Management Group, http://www.corba.org
[15] A. Eberhart, "Towards Universal Web Service Clients," Proc. Euroweb 2002, UK.
[16] "The Semantic Web Community Portal," http://www.semanticweb.org/, November 2002.
[17] S. Chatterjee , "Developing Real World Web Services-based Applications", The Java Boutique, http://javaboutique.internet.com/articles/WSApplications/
[18] Apache, "The Web Services Invocation Framework (WSIF)," http://ws.apache.org/wsif/
[19] IBM, "Web Services Security (WS-Security) version 1.0," http://www-106.ibm.com/developerworks/webservices/library/ws-secure/ , April 2002
[20] Microsoft, "Web Services Policy Framework (WS-Policy) version 1.0," http://msdn.microsoft.com/webservices/, December 2002
[21] IBM, 'Web Services Transaction (WS-Transaction) version 1.0," http://www-106.ibm.com/developerworks/library/ws-transpec/, august 2002
[22] BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, "Business Process Execution Language for Web Services (BPEL4WS) v 1.1," May 2003
[23] A. Arsanjani, B. Hailpern, J. Martin, P. Tarr, "Web Services Promises and Compromises", ACM Queue, Vol. 1 No. 1, March 2003 http://www.acmqueue.org/
[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ", Procs. ECOOP'01, vol. 2072 of LNCS:327--353.
[25] H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software", Communications of the ACM, 44(10):43--50, Oct. 2001.
[26] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns," Proc. ICSE, 1999.
[27] A. Ankolekar et al., "DAML-S: Web Service Description for the Semantic Web", Proc. 1st Int'l Semantic Web Conf. (ISWC 02), 2002.
[28] Sivashanmugam, K., Verma, K., Sheth, A., Miller, J., "Adding Semantics to Web Services Standards", Procs. of ICWS'03, Las Vegas (June 2003) pp. 395 - 401.