# Modularizing Web Services Management with AOP

María Agustina Cibrán, Bart Verheecke

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

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

**Abstract.** Web service technologi es accelerate application development by allowing the selection and integration of third-party web services, achieving high modularity, flexibility and configurability. However, current approaches to integrate web services in client applications do not provide any management support, which is fundamental for achieving robustness. In this paper we show how *Aspect Oriented Programming* (AOP) can be used to modularize service management issues in service oriented applications. To deal with the dynamic nature of the service environment we suggest the use of a dynamic aspect-oriented programming language called JAsCo. We encapsulate the management code in *aspects* placed in an intermediate layer in between the application and the world of web services, called Web Services Management Layer (WSML).

## 1. Introduction

Web services (WS) are modular applications that are described, published, localised and invoked over a network. Web services technologies accelerate application development by allowing the selection and integration of third-party web services, achieving high modularity, flexibility and configurability. However, current approaches only allow this integration by hard wiring the references to concrete web services into the client applications. As stated in [1], this leads to unmanageable applications that cannot adapt to changes in the business environment (e.g. a service that is abandoned or changed, a new service that becomes available on the market, etc). Moreover these approaches do not provide any management support, which is fundamental for achieving robustness. To deal with these issues, code has to be written manually and repeated for each service, resulting scattered in the application. We observe the need for the application to be independent of specific services.

The focus of this paper is to show how the modularization of service management issues can be enhanced by using dynamic *Aspect Oriented Programming* (AOP) [2] [3]. To deal with the dynamic nature of the service environment we suggest the use of a dynamic aspect-oriented programming language called JAsCo [4] [5]. We encapsulate the management code in *aspects* placed in an intermediate layer in between the application and the world of web services, called **Web Services**

**Management Layer (WSML)** [6]. In the next section we motivate the need for AOP and introduce JAsCo. In section 3 we show how JAsCo is ideal to modularize the management functionality of the WSML and provide some code examples. Finally, we present our conclusions in section 4.

## 2. WS Integration and Management as Crosscutting Concerns

The web service architecture is the logical evolution of object-oriented principles in a distributed context. Just as in object oriented approaches, the fundamental concepts of web services are encapsulation, message passing, dynamic building, interface description and querying. However, the distributed nature of web service applications leads to the emergence of various management concerns that are difficult to modularize using traditional software engineering methodologies.

First of all, we want to avoid hard wiring references to concrete services in the applications achieve high flexibility in the selection of services. By decoupling web services from the client application the concept of *most suitable service* is introduced. With current approaches it would be the responsibility of the application to decide which the most appropriate services are. This way, code for implementing service selection would be written at each point where some service functionality is required, resulting tangled and scattered in different places in the application. Thus, we need support for encapsulating this crosscutting code separated from the application and plug it in and out in a non-invasive way.

Moreover the selection of services also involves other management issues to be considered at the moment the services are integrated in the applications. For instance, services might need to control security, accounting, billing concerns at the time their functionality is requested. This also results in crosscutting code since the application developer would need to include this management code each time a service is requested.

Therefore, to avoid tangling the application code with service related code we identify the need for AOP. AOP states that some concerns of a system, such as synchronisation and logging, cannot be cleanly modularized using current software engineering methodologies, which leads to code duplication. To this end, AOP approaches introduce a new concept that is able to modularize crosscutting concerns, called an *aspect*. An aspect defines a set of *join points* in the target application where the normal execution is altered.

Using aspects to express the selection and management concerns as part of the WSML allows the application to remain independent of the service selection infrastructure. Moreover, we also pursue dynamism in the management of services and therefore an AOP technology that provides support for dynamic inclusion and removal of aspects is required. For this reason we introduce an aspect-oriented implementation language called JAsCo. JAsCo combines the expressive power of

AspectJ [7] with the aspect independency idea of Aspectual Components [8]. Originally JAsCo was designed to integrate aspect-oriented ideas into Component-Based Software Development [9]. However, JAsCo has some characteristics that are also useful in an object-oriented context:

- Aspects are described independently of a concrete context, making them highly reusable.
- JAsCo allows easy application and removal of aspects at run time.
- JAsCo has extensive support for specifying aspect combinations.

JAsCo introduces two concepts:

- **Aspect Beans:** specify crosscutting behaviour by defining hooks which specify *when* the normal execution of a method should be intercepted and *what* extra behaviour should be executed.
- **Connectors:** apply the crosscutting behaviour of the Aspect Beans specifying *where* the crosscutting behaviour should be deployed.

JAsCo enables the run-time plug in and out of connectors. This high flexibility and configurability is exactly what is needed for the management of web services. For more information about JAsCo we refer to [4], [5].

## 3. JAsCo Aspects in the WSML

### 3.1 Introducing WSML

In [6] we present an abstraction layer, called **Web Services Management Layer (WSML)**, which is placed between the application and the world of web services. It realises the concept of *just-in-time integration of services*: multiple services or compositions of services can be used to provide the same functionality.

Figure 1 illustrates the general architecture of the WSML. On the left side the core application resides, and if necessary, web service requests are issued to the layer. The WSML is responsible for choosing the most appropriate service or composition in a completely transparent way. This is realised by the **Selection Module** by considering different service properties. The collaboration with the **Monitoring Module** is required for this purpose as several properties of services might need observation over time.

Additional management functionality resides in the layer like traffic optimisation, billing, accounting, security, transaction, etc. The WSML is reusable in new applications and is completely configurable to avoid unnecessary overhead.

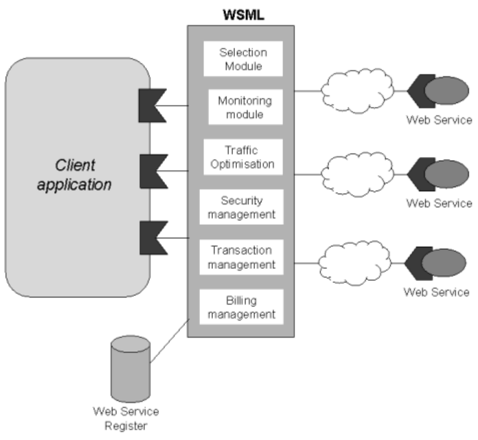

**Fig. 1.** General Architecture of WSML

The WSML has the following advantages:

- The application becomes more flexible as it can continuously adapt to the changing business environment and communicate with new services.
- Extracting all web service related code from the core application facilitates future maintenance of the code.
- Weakening the link between the application and the service enables hot swapping of services.

In the remainder of this section generic management aspects to deal with the crosscutting concerns will be presented.

## 3.2 Using Aspects for Service Redirection

Figure 2 shows how we implement the WSML using JAsCo aspect beans and dynamic connectors. A basic requirement is that hard-wiring services should be avoided. Therefore, service requests must be formulated in an abstract way at the left side of the layer and the WSML will be responsible for making the translation to a concrete service at the right side. The requests of the application are formulated in an abstract way as specified in an **Abstract Service Interface (ASI)**. This can be seen as a contract specified by the application towards the services. This way the syntactical differences between semantically equivalent services can be hidden. In order to enable this we introduce the concept of mapping schemas with sequence diagrams that unambiguously describe how the service or service composition maps to the ASI. An example of this mapping can be found in [6].

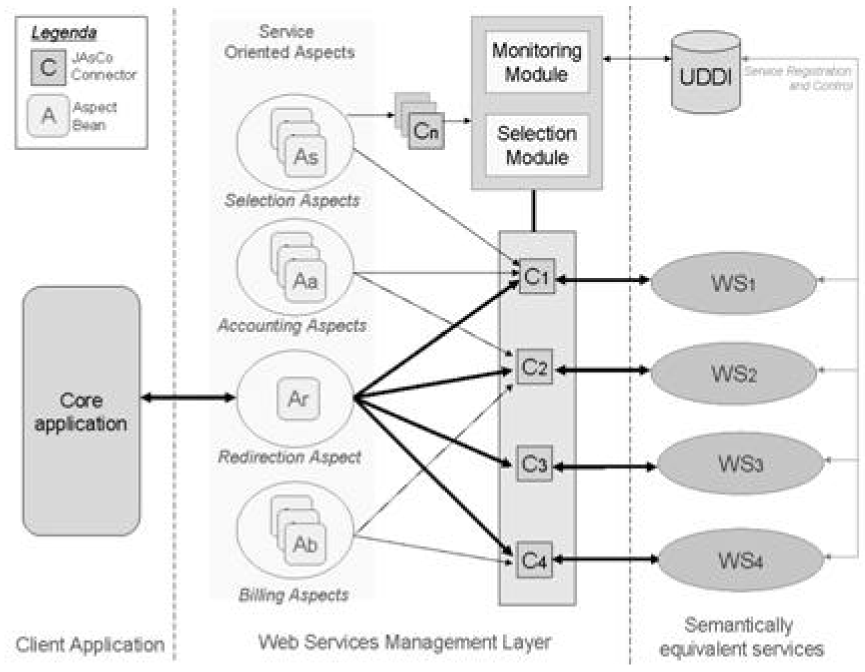

**Fig. 2.** Detailed Architecture of the WSML

To illustrate these ideas an example of a travel agency application is introduced. The application offers the functionality to book holidays online and customers can make reservations for both flights and hotels. To achieve this functionality the agency application integrates different web services. Suppose HotelServiceA and HotelServiceB are services that offer the same functionality for the online booking of hotels. Each hotel service returns exactly the same results.

Assume in the client-application a list of hotels needs to be shown to the customer. A HotelServiceInterface is defined with the following method for this purpose. `HotelList giveAvailableHotels(Date, Date, CityCode)`. At deployment time or at runtime the following two services are available: HotelServiceA provides the method: `giveHotels (CityCode, Date, Date,).` HotelServiceB provides the method: `listHotels (Date, Date, CityName).`

To make the mapping possible between the ASI and the concrete service interfaces, we make use of the aspect power of JAsCo and define an aspect in charge of redirecting the generic requests to the concrete services that will provide the functionality required. The redirection aspect defines the logic of intercepting the application requests and replacing them by a concrete invocation on a specific web service. Figure 3 shows the code for the redirection aspect. Note that this aspect is generic and does not refer to any concrete web service. The mapping to concrete web services is specified in the connectors that deploy the redirection aspect. Several connectors can exist each in charge of deploying the redirection to a concrete web service. Figure 4 illustrates the deployment of the redirection aspect. The connector HotelServiceA specifies the mapping between the ASI `giveAvailableHotels (Date, Date, CityCode)` and the particular way to invoke that functionality on the web service HotelServiceA, that is invoking the method `giveHotels (CityCode, Date, Date,)`. To communicate with HotelServiceA the GLUE library is used [10].

```
class getAvailableHotelsRedirection {
 hook RedirectionHook {
  RedirectionHook(method (Date d1,Date d2,CityCode
cc)){
   call(method);
 }

 replace() {
  specificMethod(d1, d2, cc);
 }
 abstract public List specificMethod(
           Date d1,Date d2,CityCode cc);
 }}
```

**Fig. 3.** The Redirection Aspect Bean for hotel retrieval

```
static connector getAvailableHotelsOfServiceA {
 HotelServiceAStub hotelServiceA = null;
 try {
  hotelServiceA = HotelServiceAHelper.bind();
  // the stub is instantiated by analysing the WSDL-
  // file of hotelServiceA by using the GLUE library
  }
  catch(Exception e) {  }
 getAvailableHotelsRedirection.RedirectionHook rhook =
   new getAvailableHotelsRedirection.
      RedirectionHook(Application.
         giveAvailableHotels(Date, Date, CityCode){
 public List specificMethod(Date d1,Date d2,CityCode
cc){
  return hotelServiceA.giveHotels(cc, d1, d2));
 }
}}
```

**Fig. 4.** Connector that deploys redirection aspect

Each connector encapsulates the mapping between each generic request in the application and the concrete manner to solve that request in a specific service. Thus, there will be one connector for each different request that can be invoked by the application. The WSML is responsible for the creation and management of these connectors. JAsCo allows the creation of connectors to be done dynamically. This characteristic enables the dynamic integration of new services. When the functionality of a new service has to be integrated in the application, a connector realizing the mapping for that service is created at run time. This is achieved transparently for the application.

## 3.3 Using Aspects for Service Management

As mentioned above, the layer can also deal with other management issues that need to be controlled at the application side. For instance, suppose the HotelServiceA describes a strategy for billing its use and the application wants to locally control this for auditing reasons. Suppose the service specifies that each time the method `giveHotels (CityCode, Date, Date)` is invoked, an amount of 2 euros has to be paid. We can achieve this in a non-invasive way by defining a new aspect that abstracts the logic for a "pay per use" billing strategy. Figure 5 shows the implementation of this aspect. Note that the redirection aspect is generic and can be deployed and customised for other services that adopt this billing policy. This deployment is specified as part of the connector shown in Figure 6. In this example, the billing is done when `getAvailableHotels` is invoked in the application. However, as connector `getAvailableHotelsOfServiceA` implements this method as a call to HotelServiceA, the billing is only done when this concrete service is used. Note that the hook can also be initialised with multiple functionalities provided by a web service.

```
class BillingPerUse {
  hook BillingHook {
    private int total = 0;
    private int cost = 0;

    public void setCost(int aCost){
      cost = aCost;
  }
  private void pay(){
    total = total + cost;
  }
  BillingHook(method (Date d1,Date d2,CityCode cc)) {
    call(method);
  }
after() {
    pay();
  }
}
}
```

**Fig. 5.** Billing Aspect

```
static connector getAvailableHotelsOfServiceA {
…
BillingPerUse.BillingHook billPerUse =
  new BillingPerUse.BillingHook(List
    Application.giveAvailableHotels(Date, Date,
CityCode));
  billPerUse.setCost(2);
  rhook.replace();
  billPerUse.after();
}
```

**Fig. 6.** Billing connector

The aspect BillingPerUse defines a billing template that can be reused by different services. Other more complex billing aspects can be formulated and implemented in a similar way.

This simple example illustrates that a generic library of aspects can be created to achieve high flexibility in the creation and manipulation aspects that implement other management issues.

## 4. Conclusion

In this paper we show how the use of AOP is needed to fully decouple service management concerns from the client applications. We propose to use a dynamic AOP implementation language JAsCo to enable hot-swapping and runtime management of services.

This approach has the advantage that applications become adaptable as they can easily integrate new services and dynamically accommodate to management requirements.

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 issues. We are also working on realising the hot swapping mechanism in a more intelligent way by considering service oriented rules. These rules are derived from the requirements the application specifies and are based on the non-functional properties of services.

## 5. References

[1] J. Malhotra, Ph.D., Co-Founder & CEO interKeel Inc., "Challenges in Developing Web Services-based e-Business Applications," Whitepaper, interKeel Inc., 2001

[2] Aspect-Oriented Software Development. http://www.aosd.net/

[3] Communications of the ACM. Aspect-Oriented Software Development, October 2001.

[4] D. Suvée and W. Vanderperren. "JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development". Proc. of 2nd Int. Conf. on AOSD, Boston, USA, 2003.

[5] 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, Warshaw, Poland, April 2003.

[6] B. Verheecke, M. A. Cibrán. "AOP for Dynamic Configuration and Management of Web services in Client-Applications". ICWS'03-Europe (submitted)

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. "An overview of AspectJ". In Proceedings European Conference on Object-Oriented Programming, volume 2072 of Lecture Notes in Computer Science, pages 327--353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[8] Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components. Technical Report, NU-CCS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[9] C. Szyperski. Component software: Beyond Object-oriented programming. Addison-Wesley, 1998.

[10] The Mind Electric, "The Glue Platform," 2003, http://www.themindelectric.com/glue/index.html