How to Deal with Non-functional Properties in Web Service Development*

Guadalupe Ortiz, Juan Hernández, and Pedro J. Clemente

University of Extremadura, Quercus Software Engineering Group
{gobellot,juanher,jclemente}@unex.es
http://quercuseg.unex.es

Abstract. Web Service technologies offer a successful way for interoperability among web applications. However, current approaches do not propose an acceptable method to decouple non-functional properties from Web Service implementations, leaving as a result a large amount of code scattered and tangled all over the application, thus raising problems at design, implementation, maintenance and evolution. It is the aim of this paper to describe how aspect-oriented techniques allow these properties to be easily modularized and reused. We will also analyse how information about properties can be added in the WSDL file, in order to keep clients informed of the characteristics of the service they are going to use. Finally, we will demonstrate how the client will be able to choose which optional properties have to be applied in his invocation in a transparent way, automatically generating the necessary changes in his code in a modularized and decoupled way.

1 Introduction

Web Services convey one step further in the long way that object-oriented technologies and distributed platforms have walked. However, due to the juvenility of this technology, some important points, such as non-functional property addition to the services, have not been faced yet, in spite of their being essential, considering their fast evolution to bigger and more complex services.

The addition of non-functional properties to Web Services leads to implementation changes, as these properties can actually affect various modules in the application. That is not very appropriate, since we would then have many lines of code repeated along our application with its consequent loss of time on development and maintenance. Furthermore, there are properties such as *logging*, *timing* and *security* amongst others, which are normally required in plenty of different applications; therefore, it would be desirable to be able to reuse them should they be necessary.

This problem was already faced in component-based programming [8]. Containers deal with non-functional properties [3], and there are also approaches where A*spect-Oriented Programming* (AOP) [6] is used to deal with such properties [1] [9]. Regardless of the fact that the concept of container is not contemplated by Web Service technologies in the sense of component programming, the main contribution of this

* This work has been developed with the support of CICYT under contract TIC2002-04309-C02-01

D. Lowe and M. Gaedke (Eds.): ICWE 2005, LNCS 3579, pp. 98–103, 2005.

paper is our claim that aspect-oriented techniques may be used for adding non-functional properties to Web Services. This way, we could abstract our deployment from these properties in the implementation to deal with them later. In addition, we will show how WSDL files may be modified, appending information on the properties added. In this sense, some properties are service-exclusive, that is, they do not affect the clients. However, other properties in the service, which do affect the client in the operation result or involving changes to his code, can be used by them optionally.

The rest of the paper is organized as follows: in Section 2, a case study is presented to identify the said problems. Section 3 outlines how AOP can help to solve this problem, and how AspectJ has been used and applied in Web Service development, allowing the addition of different kinds of non-functional properties in a modularized way. In Section 4, we discuss our proposal, whereas other related approaches are examined in Section 5, and the main conclusions are presented in Section 6.

2 Crosscutting Concerns in Web Services

Consider a simple example of a travel agent service, which offers four different operations: *countryAirportInformation* provides information about all the airports in a particular country; *nameAirportInformation* provides information on a given airport, *weatherAirportInformation* provides information about the weather in the airport in question; finally, *buyAirlineFlight* allows the user to buy a pre-booked flight.

The non-functional *timing* property could be added in order to calculate the time of use of two of our operations: *countryAirportInformation* and *nameAirportInformation*. This addition would cause code to be repeated and scattered all over the application, which implies not only a bad design, but also problems in maintenance and evolution. Furthermore, if we now wished to add the same property to the other operations, we would have no chance of reusability.

AOP is the answer to the problem, since it was created to design and code crosscutting concerns. AOP deals with elements that are scattered all over an implementation. As a result, we can modify these properties without influencing the rest of the code in the application. AOP is also successfully used in the Web Service domain for implementing orchestrations and reusing their interaction patterns [7].

3 Adding Non-functional Properties to Web Services

In this section we are going to show how aspect-oriented techniques may be used in order to solve the difficulties presented above. AOP describes five kinds of elements to modularize crosscutting concerns: firstly, we have to define the *join point* model which indicates the points where new behaviours could be included. Secondly, we have to define a way to indicate the *join points* in question to specify in which points of the implementation we wish to insert the new code. Next, we ought to determine how we are going to specify the new behaviour. We would then encapsulate the specified join points and their corresponding behaviours into independent units. Finally, a method to weave the new code with the original one has to be applied [4].

Non-functional properties are always added in the service. However, property additions to the service can sometimes affect the client, not only on the result of the invo-

cation, but also involving changes in his code. In this sense three different alternatives will be studied: properties which do not affect the client, properties which affect the client on the result, and those which affect the client in his codification.

3.1 Properties Which Do Not Affect the Client

The *timing* property can be modelled as an *AspectJ* aspect. This property only affects the service, as the client obtains the same information from the operation whether the time of use is being monitored or not. For our example, the property would be implemented as depicted in *Figure 1*, where we highlight *pointcut Information()*, which injects code in the execution of methods *countryAirportInformation* and *nameAirportInformation*. The corresponding *advice* shows the code to be injected.

```
public aspect TimingAspect {
pointcut Information(): execution(public * *.countryAirportInformation(..))  || execution (public *.* nameAirportInformation(..));
around (): Information(){   long T1 = System.currentTimeMillis();
                            proceed();
                            long T2 = System.currentTimeMillis();
                            long timeTaken= T2-T1;// timeTaken used for its requirements}
```

Fig. 1. Operations offered by *TravelAgentService* with an aspect modeling the *timing* property

The WSDL document is composed of different tags related to the interface, operations, parameters, ports..., of the service. The information appended, as shown, has the duty to communicate new added properties to the client. This information facilitates the properties description, the operations affected and whether they are optional or not, as depicted in *Figure 2*. The WSDL file follows the W3C standard. This is the reason why we have included that information in the *documentation* tag; consequently, any client who does not know its function will not be affected at all.

```
<documentation>    <property name="Timing">
                          <description=" Timing property : Measures the operations execution time/>
                          <optional="no">
                          <applied to>   <operation name="countryAirportInformation></operation>
                                         <operation name="nameAirportInformation></operation>      </applied to>
                    </property>
</documentation>
```

Fig. 2. WSDL documentation tag with the timing property added

With the intention of making property addition with *AspectJ* transparent to the developer, apart from replacing the *Java* compilation by the *AspectJ* weaving, we have created some additional compilation targets and a code generation process which will be invoked during service building. Therefore, the developer chooses which property he wants to apply and, in a totally transparent way, the aspect is generated and the information added to the WSDL file at compilation time. This way, the developer, when designing the application, can focus on its main functionality, adding the properties from a pool at a later stage.

3.2 Properties Chosen by the Client

We can also use the information added in the WSDL file to offer the client the possibility of choosing which properties he wants to be applied during his invocation. In

order to enable this to happen, two new processes have been included: one in the client which intercepts the outgoing SOAP message to add the information about which properties should be applied; the other one will be in the service, which intercepts the incoming SOAP message to determine which properties have been chosen by the client. On the other hand, we are going to differentiate between two cases: firstly, those properties which can be chosen without the need of modifying the client code, and secondly, those which do imply changes in the client code. Both processes are represented in *Figures 3a)* and *3b)*, respectively.

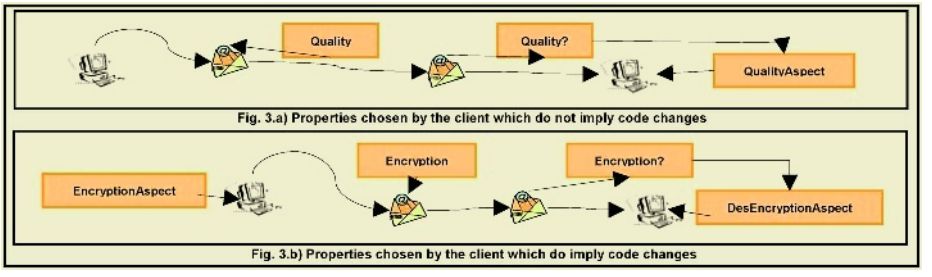

Fig. 3. Processes for the addition of non-functional properties chosen by the client

3.2.1 Properties Which Do Not Imply Changes in the Client Code

Let us consider we have a service which offers stock market quotations. The service can offer the same operation with real time or delayed quotations depending on the kind of client. The client does not need to make changes to the code, but the result may be different. Currently, the operation returns the delayed result (for regular clients), but with the *Real Time* property application, it returns the real time result (for advanced clients). The advanced client developer will include the new target related to the *Real Time* property at compilation. This target will modify the class which intercepts the outgoing message for this property to be applied. Otherwise, the class which intercepts the incoming message is included at service side to determine which properties are requested, thus the aspects will implement one behaviour or the other.

In our case study we could apply one property of this type to the operation *airportWeatherInformation,* as represented in *Figure 3a).* The operation returns the temperature when requested by ordinary clients; however, advanced clients receive a larger amount of weather information. The implementation at client side is the same, but the developer includes the *Quality* property so as to get more information.

3.2.2 Properties Which Imply Changes in the Client Code

Consider we want to provide security to our service. The *Encryption* property would imply changes in the client codification. The service currently offers its operations without encryption, but those clients who wish to have their invocations encrypted can do so. The process, shown in *Figure 3b)*, is as follows: The developer will include the encryption target at compilation time so the class which intercepts the outgoing message will include this property and the *encryption* aspect will be added. Similarly, the same changes that took place in the previous case are implemented in the service: the incoming message will be intercepted, and if *encryption* was included, the parameters will be desencrypted by the aspect at destination.

4 Discussion: Attributes and Shortcomings

The benefits provided by the use of aspect-oriented techniques are twofold. Firstly, it avoids crosscutting concerns, so service maintenance is kept simpler and the application structure well modularized. On top of that, aspects may be reused to build other services, thus diminishing developing costs and efforts, while improving the flexibility, reliability, and reusability of the application. In addition, the client is notified of the non-functional properties that affect the Web Service through the WSDL documentation. In this sense, the client can choose from the battery of properties offered by the service which ones should be applied during his invocation, and all the necessary code will be generated automatically in a totally transparent way.

Table 1. Performance measurements for the addition of non-functional properties

	Timing		*Quality*		*Encryption*	
Metrics	O	AOP	O	AOP	O	AOP
ART (ms)	5142,5	5393,3	2225,571	24880,5	848,75	886,5
ACN	0	1	0	2	1	1
PMM (%)	45	0	60	0	30	0
EIN	1	0	0	0	1	0

Finally, *Table 1* shows the performance metrics we have measured on the example implemented with (AOP) and without (O) AOP techniques. To start with, the *average response time* (ART) is quite similar in both implementations, although the AOP time is slightly longer, it is not a big price to pay for the improvements. Secondly, the *number of classes added* (ACN) is bigger when using AOP, as we implement the properties in aspects. Besides, no *method* has *to be modified* at all when using aspects (PMM), whereas, when no aspects are used, every method which will be offered as a new operation will undergo modification, with a high *percentage* of lines to be added. Finally, new *external invocations* (EIN) have to be added if we do not use AOP. For all these reasons, our proposal proves to be rather efficient at performance as response time is not wasted while improving modularity, reusability and maintenance.

5 Related Work

Although there are undoubtedly important infrastructural issues concerning Web Services, there seems to be little discussion on how to add non-functional properties to them.

Firstly, we can distinguish one contribution from S. Göbel *et al.*, where non functional properties, such as security, accuracy or other service quality-related ones, are specified in component models by using aspects [5]; we walk one step ahead using this technology along with Web Services.

Secondly, MB. Verheecke et al. suggest the use of a dynamic aspect-oriented language called *JAsCo* for decoupling services from the application that invokes them [10]. They focus mainly on the client side; in contrast, our proposal uses a general use aspect-oriented language and is mainly centred on the server side.

Finally, we can also mention a paper that concentrates on a new language based on XML, AO4BPPEL, being one aspect-oriented extension for BPEL [2]. In contrast with our proposal based on languages with wide applications support, they need a new weaver for the proposed language, which is not available on the Web nowadays.

6 Conclusions

The results obtained in this study show how AOP is really useful in order to avoid crosscutting in Web Service development. In particular, AspectJ has been used for dealing with concerns which crosscut Web Service implementation, while improving the reusability of non-functional properties in the development of different services.

One of the main advantages of our proposal is the possibility of adding these properties without modifying the service code, as well as adding some information about them in the WSDL file, without hindering the client who disclaims the use of aspects tags. The client who is aware of the usefulness of this information can choose the properties which will be applied during his invocation. The necessary code would be generated in a totally transparent way both in the service and in the client's side.

References

1. Bonér, J. What are the key issues for commercial AOP use: how does AspectWerkz address them? Proc. 3rd Int. Conf. AOSD, ACM Press, Lancaster, UK, 2004
2. Charfi, A., Mezini, M. *Aspect-Oriented Web Service Composition*, Proc. 2004 European Conference on Web Services (ECOWS 2004), Erfurt, Germany, September, 2004.
3. Duclos, F, Estublier, J, Morat, P: Describing and using non functional aspects in component based applications. Proc 1st Int. Conf. AOSD, ACM Press, Enschede, The Netherlands, 2002
4. Elrad, T., Aksit, M., Kitzales, G., Lieberherr, K., Ossher, H.: *Discussing Aspects of AOP*. Communications of the ACM, Vol.44, No. 10, October 2001.
5. Göbel, S., Pohl Cristoph, Röttger, S., Zschaler, S. The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. Proc. 3rd Int. Conf. AOSD, ACM Press, Lancaster, UK, 2004
6. Kiczales, G. *Aspect-Oriented Programming*, Proc. ECOOP, Jyväskylä, Finland, June 1997.
7. Ortiz G., Hernández J., Clemente, P.J: *Web Service Orchestration and Interaction Patterns: an Aspect-Oriented Approach*, Proc. 2nd. ICSOC. New York, USA, November 2004.
8. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. Addison-Wesley (1997).
9. Tal Chen, Joseph (Yossi) Gil. AspectJ2EE=AOP+J2EE: Towards an Aspect Based, Programmable and Extensible Middleware FrameWork. Proc. ECOOP, Oslo, 2004
10. Verheecke, B., Cibrán, M.A.: AOP for Dynamic Configuration and Management of Web Services. Proc. Int. Conf. on Web Services, (ICWS-Europe'03) Erfurt, Germany, (2003)