

Grundy, J.C., Panas, T., Singh, S., Stoeckle, H. An Approach to Developing Web Services with Aspect-oriented Component Engineering, In Proceedings of the 2nd Nordic Conference on Web Services, 2003.

An Approach to Developing Web Services with Aspect-oriented Component Engineering

John Grundy^{1,3} Thomas Panas² Santokh Singh¹ Hermann Stöckle¹

¹Department of Computer Science and
³Department of Electrical and

Electronic Engineering, University of
Auckland

Private Bag 92019, Auckland, New Zealand
{john-g|santokh|herm}@cs.auckland.ac.nz

²Department of Computer Science
Växjö University

Vejdes Plats 7, 351 95 Växjö,
Sweden

thomas.panas@msi.vxu.se

Abstract Web services have become a popular new technology for describing, locating and using distributed system functionality. However, existing web service development approaches lack aspect-based development support for distributed components. We describe the application of Aspect-Oriented Component Engineering to web service development. This includes grouping web service operations into components and characterising the cross-cutting functional and non-functional aspects of these components such as transaction support, distribution technology, persistency, performance and reliability, security, resource utilisation, collaboration support and so on. Web services are described with an extended, aspect-oriented description language and are indexed and located using these aspect extensions. Aspect descriptions are used to statically and dynamically validate located services and to integrate them with client components. We describe an example of applying this approach to a highly distributed sample application.

Introduction

Most new distributed systems now use internet technologies as a fundamental part of their architecture. This has led to demand for an open, stable, scalable software infrastructure for the development of applications for E-Business [15]. Up until recently most distributed system infrastructures and technologies, such as CORBA, DCOM, EDI and XML over TCP/IP, have provided some useful techniques for abstracting remote component interfaces and supporting cross-organisational communication [2,9, 12, 15]. However most have lacked the ability to work over a wide variety of internet services with security constraints, have lacked adequate

dynamic queryable descriptions and binding services, have used proprietary solutions, or have limited cross-platform or cross-language support features (including complex data structure representation).

One solution to these problems has been the development of web services [3, 8, 15, 14]. These are basically remote component services described, located and accessed using a set of open standards from the W3C. This new concept allows very promising possibilities of heterogeneous application integration over the internet [3]. Web services have quickly become popular in large part because they build on a well known and widely accepted meta language, XML. They can provide a basic communication infrastructure on which existing remote object systems, such as DCOM or CORBA, can operate, by using HTTP as a de facto Web Service message carrier. They provide a simple, standardised mechanism for describing web services (“Web Service Description Language”, WSDL), dynamically locating web services (“Universal Description and Discovery Interface”, UDDI), and co-ordinating cross-system processes (“Business Process Execution Language for Web Services”, BPEL4WS).

However, web services are still an immature technology. Many questions, regarding their performance, security or interoperability, are yet not answered [5, 14, 11]. In addition, most web service-based systems are currently designed using conventional object-oriented analysis and design approaches. During development of a number of distributed systems we have found that such design approaches do not adequately help developers to capture, reason about and encode higher level component capabilities and are especially poor with respect to addressing issues cross-cutting component services [1, 2]. We developed Aspect-oriented Component Engineering (AOCE) to address these concerns for conventional distributed component-based systems [1].

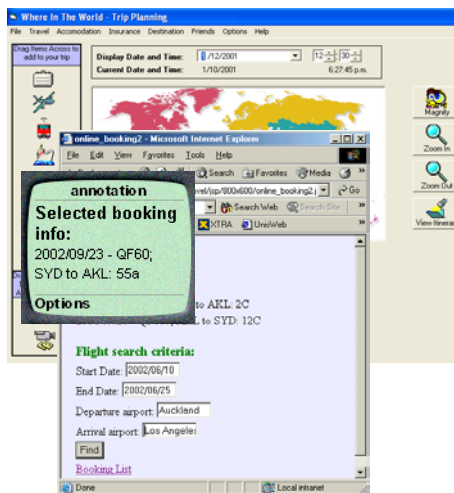
In this paper we report on our attempts to apply AOCE to the design and development of web service-based distributed systems, to try to capture and use more knowledge about web service-implemented distributed components at both design-time and run-time. We motivate this work with an example of a highly distributed application, a travel planner, and outline the AOCE methodology. We then describe how AOCE concepts can be used when designing web service-based components to describe their cross-cutting concerns and to reason about component interactions. We describe how an extended form of WSDL, AO-WSDL, can be used to capture cross-cutting web service component information and how an extended form of UDDI, AO-UDDI, can be used to dynamically discover and integrate web service components by leveraging aspect-oriented information in AO-WSDL descriptions. We summarise with the contributions of this work and some directions for future research.

Motivation

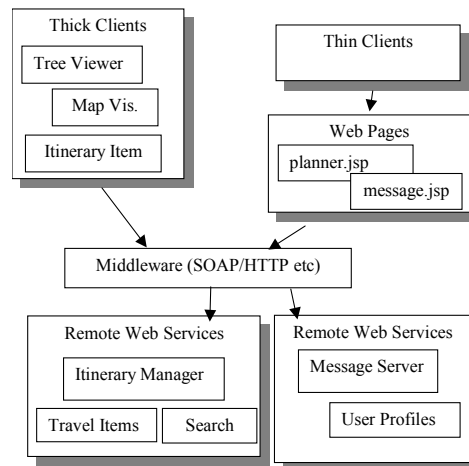
Web services are self-describing components that can discover and engage other web services or applications to complete complex tasks over the internet [3, 8, 15]. Historically, client server computing is designed around the intranet model where software development is based on components operating inside an organisation’s

firewall, using distributed object technologies like DCOM, CORBA, J2EE, and EJB. Web services primarily use HTTP as a transport mechanism and XML as the inter-application communication protocol. Technically, a web service is a XML-based call to software components on distributed servers. In the centre of the web service technology is XML as a universal data and message structuring and encoding language. Actual inter-application message exchanges are transmitted through SOAP (Simple Object Access Protocol). Web services can also be described by XML documents, typically by the Web Services Description Language (WSDL) format. Web services can be registered with repositories and located by and bound to by other web services at run-time. The detection of the web services is by standard repository services, which are typically based on the Universal Description, Discovery and Integration (UDDI) language [16]. Several implementations of web services exist, the most commonly used being J2EE and .NET [3].

As an example of a system that could be built with web services, consider a collaborative travel planning application, used by customers and travel agents to make travel bookings [2]. Examples of the user interfaces provided by such a system are illustrated in Figure 1 (a), and some software components composed to form such an application are illustrated in Figure 1 (b).



(a) Example travel planning application interfaces



(b) Example travel planner components

Figure 1. Example component-based application.

In this example a distributed component-based system has been built by composing a set of software components to provide the range of facilities required: travel itinerary management; customer and staff data management; system integration with remote booking systems; and various user interfaces, including desktop, web-based and mobile device interfaces. Some components are quite general and highly reusable e.g. map visualisation, customer data manager, chat and email message clients and server, and middleware and database access components. Others like the travel itinerary

manager, travel item manager, travel booking interfaces and integration components, are more domain-specific. Many facilities, such as locating travel items, booking items and payment will be provided by remote third-party systems. These will likely be widely distributed and systems may provide quite different interfaces and protocols e.g. different travel item formats, searching facilities, booking business processes and so on.

Ideally we would like to design and develop components that can be described, dynamically located and integrated into this system as needed. This requires in-depth knowledge about not only the component type interfaces but aspects of their non-functional behaviour as well. Earlier versions of this travel planner prototype that we have developed used older component technologies which were not able to support this. While web services provide a basic infrastructure that allow this, current design techniques and web services description and registry technologies do not fully support identification and use of cross-cutting concerns on web service-implemented software components [13].

To show that our methodology is platform and language independent, besides using Java's J2EE, we have also used Microsoft's .NET technology to implement the Travel Planner system. We used Visual Studio .NET and C# to build our web services and clients to consume them. We applied AOCE techniques to implement the travel planner clients and also all the web services for Flights, Car Rentals, Hotel Room Reservation and Payment in our collaborative travel planning application. The AOCE methodology was used to develop this system from inception to implementation and subsequent maintenance. We have refactored crucial parts of the travel planner implementation several times and discovered that refactoring was a much easier and less stressful task as compared to doing it without the AOCE methodology. This could be attributed to the fact that using aspects and components in our designs and implementations brought about greater consistency and coherency in our approach and this gave us more control and increased understanding in the web services system that we were developing.

Aspect-oriented Component Engineering

When building such an application from parts i.e. when using component-based development, developers typically assemble components that have been identified and built using "functional decomposition": organising system data and functions into components based on the vertical piece(s) of system functionality these support. However, many systemic features of an application end up cross-cutting, or impacting, on many different components in the system. For example, things like user interfaces, data persistency, data distribution, security management and resource utilisation all impact a wide variety of components and some of the component methods and state. Some components provide functionality relating to these system features, others require it from other components in order to operate. We use the term "component aspects" to describe these more horizontally-impacting perspectives on software components in a system [1].

We developed Aspect-Oriented Component Engineering (AOCE) to help developers engineer better software components [2]. *Component aspects* are broad categories of annotations we use to describe systemic system properties that components provide functions for or require functions from other components. Examples of component aspects (which we refer to just as "aspects" from here) include user interface, distribution, transaction processing, security, persistency, configuration and collaborative work support facilities. *Aspect details* describe various systemic properties under each aspect category that some components *provide* and that others *require*. For example, one component may provide a button panel (user interface aspect detail) another component may require to extend (e.g. to add its own buttons). One component may provide event broadcasting support, which another requires to do distributed communications. Each aspect detail has one or more *aspect detail properties* which further characterise it e.g. event transfer rate, memory usage size, kind of user interface affordance, synchronous vs. asynchronous group editing, and so on. Aspect detail properties may be single-valued or specify an acceptable value-range constraint. Component aspect details may overlap e.g. marshalling for persistency and distribution, feedback for user interface and collaborative work. Several component functions may be impacted by the same aspect detail and a single function may be impacted by multiple details.

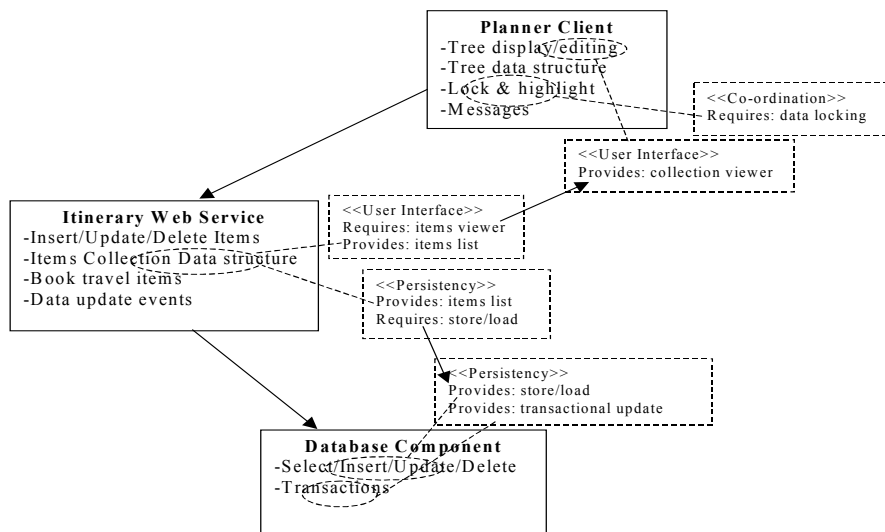


Figure 2. Concept of "Component Aspects".

Figure 2 illustrates this concept for some of the components in our web service-based travel planner application. The travel planner client provides a tree viewer as the main user interface and it also requires work co-ordination support (locking data items across multiple users). The travel itinerary web service requires a user interface to display and edit its items collection data structure. It also requires data persistency support. The travel itinerary web service provides a data structure to both render and store, and generates data update events. A database component provides data storage and transaction co-ordination support.

Each of these “component aspects” that impact parts of a software component can be categorised into “aspect details” the component provides or requires. Each aspect detail may be constrained by “aspect detail properties” that capture functional or non-functional constraints relating to these horizontal perspectives on the component’s functionality. For example, the itinerary manager may specify it requires a component providing data storage of a certain speed e.g. 1000 insert() and update() functions must be supported per second. The kind of awareness supported by the tree viewer might be specified e.g. highlight of changed items. Components providing security may indicate the kind of authentication or encryption used. Components requiring memory management facilities may indicate the upper bounds of resources they use, performance they require or concurrency control techniques they need enforced.

Developing Web Services with AOCE

We have been applying AOCE to the development of web service-based distributed systems to improve web service component design, description and dynamic location and integration.

Figure 3 shows how AOCE is used in this context.

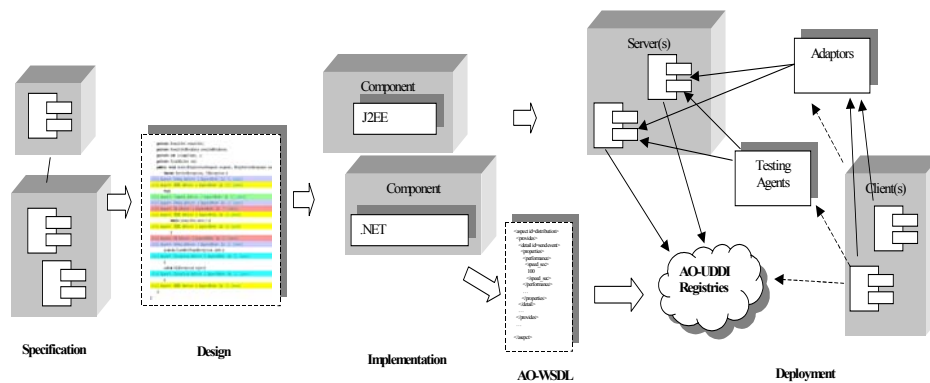


Figure 3. Using the Aspect-oriented Web Services approach.

Component specifications are translated into designs, which include aspect-based annotations of design diagrams in UML-based CASE tools, such as Rational Rose™ [1] and detailed component interface specifications in development tools, such as our Aspect-jEdit [10]. When implementing web service components in Aspect-jEdit we capture information about cross-cutting concerns, such as distribution, security, transaction processing and resource utilisation. We encode this information in an extended form of WSDL, which we call Aspect-oriented WSDL, or AO-WSDL. This describes component interface types as well as provided and required services of a component relating to cross-cutting aspects, capturing both functional and non-functional characteristics of the web service component relating to the aspects. Within the extension of Aspect-jEdit that is currently under development, aspects will be

automatically detectable. Our work relies on syntactical and semantical pattern detection to detect and visualize functional/non-functional, reusable/non-reusable and inline/outline aspects [17] automatically.

Implemented web service components are then deployed in servers and advertised for other components to connect to using an extended UDDI registry, that we call Aspect-oriented UDDI, or AO-UDDI. This extended web service component registry indexes aspect characterisations associated with our aspect-oriented web service components and allows clients to make queries for components meeting aspect constraints as well as type information. After the services are available on the web, the client can connect to a server and use the services provided. For this, the client needs first to retrieve information from the UDDI, which is a repository service where the client can allocate web services. As the UDDI carries now additional aspect information, which provides the client with further information about functional and non-functional crosscutting properties of the services. The additional information might be especially useful for agents that try to allocate similar services but with e.g. the highest reliability or performance. Automated testing agents may be deployed to check these advertised properties of an aspect-oriented web service. However, not every clients can talk to every server and suitable adaptors to the web service may need to be located e.g. to translate between CORBA and SOAP protocols, to translate between different SOAP message sets and so on.

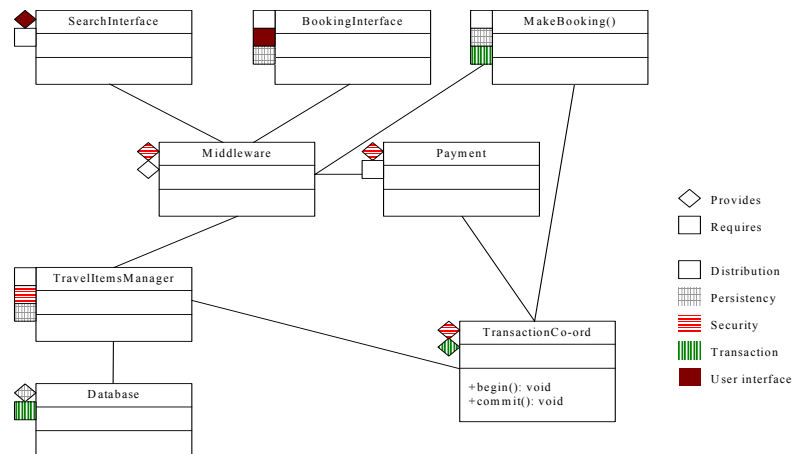


Figure 4. An example of an AOCE component design.

We can apply this development approach to the design of our travel planner application. For example, Figure 4 depicts an AOCE UML class diagram for the travel planner system. Each component is depicted as a traditional UML class. The different types of patterns in the boxes indicate the different aspects and also whether the aspects provide or require crosscutting information from another component. A square box indicates that the aspect requires the crosscutting information while a diamond shaped one provides it for another component. The figure shows the aspect crosscutting over the system, with the functional and non-functional properties attached to each visual component. This is a benefit in order to match components according to their requirements. This approach differs from traditional AOP in the

sense that the cross-cuttings are not separated out into own modules. However, this can be performed in a next step, where a second view of the system is created, indicating components and aspects separately. Therefore Figure 4 is a good view to match web-services, while a second aspect-component view would be beneficial to perform changes on the aspects.

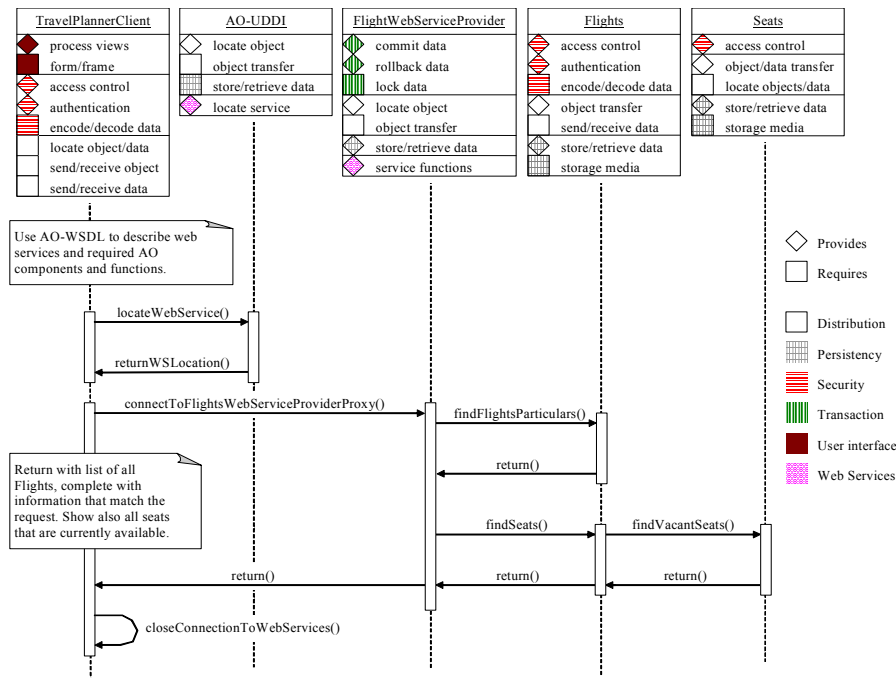


Figure 5. AO-OOD Sequence Diagram for locating seats in our Travel Planner.

Figure 5 shows an example of an AO-OOD Sequence Diagram for the Travel Planner. It describes the sequence of events for searching for seats that match given criteria in the LocateSeats component. The functions involved are clearly described together with the aspects embedded in the objects. Again, the different types of patterns in the boxes indicate the various aspects and also whether the aspects provide or require crosscutting information from another component. The shapes, patterns and colours in Figure 5 have the same meaning as in Figure 4, this consistency between different diagrams can enable a quicker understanding of the content and also ease to find faster the desired aspects or have an overview in more complex diagrams (compared to a pure textual representation of the aspects). Moreover it helps to cut down the amount of textual information in diagrams and is therefore more compact. The colour can indicate the importance of aspects, as for example security is depicted in Figure 4 and Figure 5 in red horizontal lines. A square box indicates that the aspect requires the crosscutting information while a diamond shaped one provides it for another

component. Besides different patterns, the aspects are also represented by different colours, for example the aspects for security are depicted as red horizontal lines. During implementation we sieved out the aspects that were involved so that we can identify and isolate these cross-cutting modular units in the objects. This enabled us to make the code more consistent, understandable and at the same time address the issue of crosscuts “tangling” in designs and code.

Describing Web Services with AO-WSDL

In order to support the better characterisation of web services and use this information to support easier and more dynamic, automated systems integration we need to extend WSDL to AO-WSDL. While WSDL characterises the data, operations, events and reflective information of a web service, AO-WSDL needs to carry additional information for functional and non-functional aspects. Functional characteristics of web service components describe what operations/data provide or require what system facilities, e.g. distribution, persistency or security. Non-functional characteristics of web services describe constraints on the data/operations provided by components, e.g. reliability or performance. Additionally, AO-WSDL contains composition information in order to aggregate web components to form useful cross-system business transaction processing support. We have used the extension mechanisms built into WSDL to add additional descriptive support for aspect-oriented web service components.

AO-WSDL can be automatically generated for our web services developed using AOCE. For example, from the AO-design diagrams in the previous section, the relevant aspect information can be deduced and AO-WSDL descriptions for each software component generated. These would include the aspect-encoded characteristics of components as described in the WS-AOCE design diagrams. For the travel items (itinerary) manager, the standard WSDL descriptions of the web service’s data, messages and ports would be encoded, along with aspect characterizations of the web service e.g. distribution support provided (via its web service interface and associated middleware); security and transaction management required (from the web service middleware and transaction co-ordinator service respectively); and persistency required (from the database component). As these AO-WSDL specifications for web service components follow clearly defined formal semantics, they allow for automatic searches for any given aspects, aspect details and properties of the services advertised.

An example of an AO-WSDL specification from our travel planning application is shown in Figure 6, defining an itinerary manager component with various properties, operations and event support. In this example three aspect characterisations of the web service are shown on the right hand side. Persistency (low-level) support it requires from a database component includes typical “DataManager” support, the web service operations impacted by persistency requirements are identified e.g. findItinerary, addItinerary, updateItineraryItem etc, and a performance constraint has been specified (the persistency operations need to complete in under 100ms). A transaction (medium-level) policy is required, and its characteristics include the impacted web service operations and transaction

demarcation over these operations. A domain-specific “booking” aspect is also required, a high-level aspect characterisation.

Locating AO-Web Services

The web services UDDI registry provides a way of indexing categories of web services and web service providers for remote querying and web service location. WSDL descriptions can be used both as a partial basis for organising and indexing web services and for providing communications-level descriptions of located web services to requesting clients. When locating a web service using UDDI we need to utilise AO-Web Service component information in several ways.

<pre> <component name="Itinerary Management"> <services name="" /> <!-- no web services implementing this component /> <components name="" /> <property name="caching"> <value type="boolean" /> <getter operation="getCaching" /> <setter operation="setCaching" /> </property> ... <operation name="findItinerary" style="rpc"> <arg name="ID" style="in" type="LongInt" /> <arg name="itinerary" style="out" type="itinerary:ItineraryData" /> </operation> <operation name="updateItinerary" style="message"> <arg name="ID" style="in" type="LongInt" /> <arg name="itinerary" style="in" type="itinerary:ItineraryData" /> <exception name="InvalidUpdate" message="itinerary.InvalidUpdate" /> </operation> ... <aspects namespaces="www.travelplanner.com/aspects/namespaces/itinerary"> <aspect name="ItineraryData" detail="itinerary:ItineraryDataManagement" type="provided"> <impacts operations="all" /> </aspect> </aspects> </pre>	<pre> <aspect name="Persistence" detail="common:DataManager" type="required"> <impacts operations="findItinerary/addItinerary..." /> <property name="Performance" type="common:OperationSpeed"> <common:lessThan units="ms">100</lessThan> </property> </aspect> <aspect name="TransactionSupport" detail="common:TransactionsRequired" type="required"> <impacts operations="findItinerary/addItinerary..." /> <property name="TransactionScope" type="common:TransactionDemarcation"> <common:transactionState>IN_TRANS</transactionState> </property> </aspect> <aspect name="BookingManager" detail="booking:TravelBookingManager" type="required"> <impacts operations="addItinerary/updateItinerary..." /> <property name="BookingCommittalApproach" type="booking:BookingCommittal"> <booking:BookingCommittal value="BTP" /> </property> <property name="Timeout" type="booking:Timeout"> <booking:Timeout units="days"> <max>5</max></booking:Timeout> </property> </aspect> </pre>
--	--

Figure 6. Example of AO-WSDL based component description.

Firstly, the set of appropriate web components that are provided by a service that we may wish to interact with. Secondly, determine the provided service characteristics, allowing searches to be specified with additional query conditions using aspect information. For example, a web service client could specify desired performance, security, communications technology, transaction processing, and domain-specific support provided by web components. The aspectual extensions in the AO-WSDL and AO-UDDI are used to match the services and service characteristics being looked for by a prospective web service client. These compatibility checks and searches can be automated more easily in our systems. Finally, located web components that require other component functionality in order to operate can cause additional web service queries to be run in order to provide the original requesting client with a set of components that as a composite will fulfill their needs.

As an example, consider that when a travel item provider component is located it may require loosely-coupled BTP-style transaction support, LDAP authentication and payment authorisation components in order to operate. The travel item provider e.g.

an airline flight booking service will specify not only required remote services (transaction coordinator, authentication and payment authorizer) but desired and/or required characteristics of these services via encoded AO-WSDL information. For example, a BTP-compliant transaction coordinator may be wanted; an authentication server using a specific security encryption protocol for passwords; and a payment authorizer that takes less than 15 seconds to complete payment authorization requests. Some available web services that can be found in a conventional UDDI registry may meet these characteristics but some may not. In addition, some services may need “adaptors” in order for the travel item provider to interact with them e.g. transforming SOAP messages between different protocols that impact characteristics (performance, security, transactional behaviour, resource utilization and so on).

AO-UDDI registry queries for suitable components, based the specified required aspects this travel item provider advertises, can be located by subsequent queries. Located services can have their AO-WSDL descriptions obtained and where necessary, validation agents and/or suitable adaptors located and used to test the services and integrate the requesting client with the newly discovered web service components. All these can also be automated because our aspects are well defined and follow formal semantics that are consistently used throughout our Web Services systems, AO-WSDL documents and AO-UDDI.

Summary

We have described how aspect-oriented component engineering can be applied to the design, characterisation, location and integration of web service-based software components. Web service-based component architecture designs are annotated to capture richer descriptions of the web services, particularly their non-functional characteristics relating to component cross-cutting concerns (aspects). We have extended the web service description language to allow capturing of this information in AO-WSDL XML documents. These extended descriptions can be indexed by AO-UDDI registries and the aspect information used to assist better location, testing and integration of web service components. The combination of AO-WSDL and AO-UDDI used in conjunction with web services that were developed using AOCE techniques supports richer, clearer and more superior web services systems as compared to those built without this technique. We are currently working on tool support for AO-web services development in both design tools with an extended UML modelling approach and in implementation tools using an extended Java development environment which generates AO-WSDL descriptions of web service component implementations.

References

1. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 6, December 2000, pp. 713-734.

2. Grundy, J.C. Mugridge, W.B. Hosking, J.G. and Apperley, M.D. Tool Integration, Collaboration and User Interaction Issues in Component-based Software Architectures, in Proc. of TOOLS Pacific '98, Melbourne, Australia (24-26 November 1998), IEEE CS Press.
3. Hansen, J.J. .NET versus J2EE Web Services: A comparison of approaches, Web Services Architect, January 2002.
4. Kiczales et al, Aspect-oriented Programming, in Proc. of the 1997 European Conf. on Object-Oriented Programming, Finland (June 1997), Springer-Verlag, LNCS 124.
5. Litoiu, M. Migrating to Web services - latency and scalability. In Proceedings Fourth International Workshop on Web Site Evolution, IEEE CS Press, 2002, pp.13-20.
6. McKinlay, M., Tari, Z.. DynWES - a dynamic and interoperable protocol for Web services. In Proceedings of the Third International Symposium on Electronic Commerce, IEEE CS Press, 2002, pp.74-83.
7. Mezini, M. and Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development, in Proc. of OOPSLA'98, Vancouver, WA (Oct. 1998), ACM Press, pp. 97-116.
8. Microsoft Corp, Microsoft .NET™, www.microsoft.com/net/, February 2003.
9. Mowbray, T.J. and Ruh, W.A., Inside Corba, Addison-Wesley, 1997.
10. Panas, T., Karlsson, J. and Högberg, M. Aspect-jEdit for Inline Aspect Support, In proceedings of the 3rd German Workshop on Aspect Oriented Software Development, Technical Report of the University of Essen, March 2003.
11. Piccinelli, G., Emmerich, W., Zirpins, C., Schutt, K. Web service interfaces for inter-organisational business processes an infrastructure for automated reconciliation. In Proceedings Sixth International Enterprise Distributed Object Computing Conference, IEEE CS Press, 2002
12. Sessions, R. COM and DCOM: Microsoft's vision for distributed objects, Wiley, 1998.
13. Stearns, M. and Piccinelli, G. Managing interaction concerns in web-service systems, In Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, Vienna, Austria, July 2002, IEEE CS Press.
14. Tilley et al, Adoption challenges in migrating to web services, In Proceedings of the Fourth International Workshop on Web Site Evolution. IEEE CS Press, 2002, pp.21-29.
15. Wiedemann, M. Web Services and collaborative commerce. Information Management & Consulting, vol.17, no.3, Aug. 2002, pp.57-60.
16. Zhang, L.J. Li, H., Chang, H., Chao, T. XML-based advanced UDDI search mechanism for B2B integration, In Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, IEEE CS Press, 2002, pp.9-16.
17. Panas, T. and Andersson, A. and Assmann, U. The Editing Aspect of Aspects. In I. Hussain, editor, Software Engineering and Applications (SEA 2002), Cambridge, USA, November 2002, ACTA Press.